LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
Il6r
no. 511-516
cop. 2

510.84
IL6N
no.516
cop.2

Report No. UIUCDCS-R-72-516

TRANSFORMATIONS ON LOOP-FREE PROGRAM SCHEMATA

by

Nurit Bracha

THE LIBRARY OF THE
JUL 3 1972
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

June 1972

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Digitized by the Internet Archive
in 2013

http://archive.org/details/transformationso516brac

Report No. UIUCDCS-R-72-516

TRANSFORMATIONS ON LOOP-FREE PROGRAM SCHEMATA

by

Nurit Bracha

June 1972

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

This work was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, June 1972.

TRANSFORMATIONS ON LOOP-FREE PROGRAM SCHEMATA

Nurit Bracha, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1972

A formal theoretical approach toward code optimization is considered. A program schema that models loop-free programs is presented and a complete set of equivalence preserving transformations on loop-free programs is found. A scheme for optimization is provided in which a sequence of these transformations is applied to get an optimal code. These results are extended to the model of loop-free programs which assumes that certain types of algebraic laws hold among the operators, and also to the case in which the tests are Boolean functions of elementary tests.

ACKNOWLEDGEMENT

The author wishes to express her appreciation and gratitude to Professor David E. Muller for his guidance and supervision of this thesis. The author would also like to express thanks to the Department of Computer Science and the Computing Services Office, University of Illinois, for their support of her graduate studies.

Finally the author wishes to thank her family for their constant encouragement and moral support.

TABLE OF CONTENTS

		Page
1.	INTRODUCTION	1
2	THE PROGRAM SCHEMA	4
3	EQUIVALENCE OF PROGRAM SCHEMATA	10
4.	TRANSFORMATIONS ON PROGRAMS	18
5.	OPTIMIZATION	68
6.	ALGEBRAIC TRANSFORMATIONS	78
7.	LOGICAL TRANSFORMATIONS	91
8.	OPTIMIZATION UNDER LOGICAL TRANSFORMATIONS	101
9.	PROGRAM SCHEMATA WHICH ALWAYS HALT	114
REFERENCES		120
VITA		121

1. INTRODUCTION

In many cases it is desirable to have a compiler of a programming language produce an output code which is economical with respect to some cost criterion such as program size or program speed. Optimizing a given program is usually done by an application of transformations to some intermediate language representation of the program. There are many possible transformations that can be applied and compilers which use transformations have been built (7,2). Ideally, a theory of code simplification should provide a machine independent mechanism for reducing a given program to an equivalent program which is in a simplest (in some sense) possible form.

The existence of algorithms for simplifications is closely connected with the question of the solvability of the equivalence of programs. If the equivalence problem was solvable, then an algorithm for reducing a program to a simplest form would exist in principle. It can be shown that for almost any reasonable notion of equivalence between computer programs, the question of equivalence of pairs of programs is not partially decidable (8). There is no effective procedure for determining whether or not two programs are equivalent. Therefore in general, we can not find a finite collection of equivalence preserving transformations so that any pair of equivalent programs can be transformed one into the other by applying a finite sequence of these transformations.

Despite the undecidability of the theory in general, positive theoretical results can be obtained in many cases and efforts have been

made to isolate subclasses of programs for which the equivalence problem is decidable (8, 9), and to find complete sets of equivalence preserving transformations which can be applied to decidable subclasses of programs (1).

Aho and Ullman (1) considered a type of program schema that models straight line code. For this case they found a complete set of equivalence preserving transformations and showed that these transformations can be applied to get an optimal code. They extended their results to cases in which certain types of algebraic laws are assumed.

The purpose of this thesis is to consider a program schema that models loop-free programs and to extend Aho's and Ullman's results to this case. For this subclass of programs the equivalence problem is decidable (9).

In Chapter 2 the model of loop-free program schema is presented. A program schema will represent a family of computer programs in the sense that if the operator and test names are given a particular interpretation the schema becomes a program that can be executed by an idealized computer.

In Chapter 3 the notion of equivalence of loop-free programs is defined and in Chapter 4 a set of equivalence preserving transformations is presented. This set extends Aho's and Ullman's set of transformations. Also their representation of programs by directed acyclic graphs is used. The set of transformations presented is shown to be complete, i.e. two programs are equivalent if and only if they can be transformed one into the other by a sequence of these transformations.

In Chapter 5 a schema for optimization is provided in which a sequence of the transformations is applied to get an optimal code.

Chapter 6 extends the results of the previous chapters to the model of program schemata that assumes that a set of algebraic laws holds among the operators. It is shown that in this case two programs are equivalent if and only if they can be transformed one into the other by a set of topological and algebraic transformations. It is also shown that certain types of local optimization techniques can be considered as algebraic identities that hold among operators, operands and constants.

In Chapters 7 and 8 the results are extended to the model of program schemata in which the tests are Boolean functions of elementary tests. It is proved that in this case two programs are equivalent if and only if they can be transformed one into the other by a set of topological and logical transformations. Chapter 8 provides a scheme for optimization that uses topological and logical transformations.

In Chapter 9 the subclass of programs which always halt is considered. For this subclass the equivalence problem is decidable (8, 9) although membership in this class is not (8). It is shown that the procedure for optimizing loop-free programs might be used to optimize programs which always halt. Certain transformations that are known to improve the code of programs with loops are shown to be equivalent to sequences of transformations on loop-free programs.

2. THE PROGRAM SCHEMA

Let Σ be a countable alphabet of variable names,

Θ a countable set of operator names,

T a countable set of test names.

statements are of three types:

(i) assignment statements

$$k \quad A \leftarrow \theta\, B_1 \ldots B_r$$

$A, B_1, \ldots, B_r \in \Sigma$

k is a numeral which is optional, and is the address of the statement.

θ is an r-ary operator name, $\theta \in \Theta$.

The variable A is assigned a new value, which depends on the current values of $B_1, \ldots, B_r$ and on the unspecified operator θ.

We say that A is defined by this statement and $B_1, \ldots, B_r$ are referenced by this statement.

(ii) test statements

$$k \quad t(C_1, \ldots, C_r)\ k_1, k_2$$

$t \in T$, $C_1, \ldots, C_r \in \Sigma$

k, k_1, k_2 are numerals. k is optional. k_1, k_2 may be equal.

Control goes to the statement with the prefix k_1 if the predicate $t(C_1, \ldots, C_r)$ is true, otherwise to the statement with the prefix k_2.

k_1 and k_2 are called transfer addresses. We say that $C_1, \ldots, C_r$ are referenced by this statement.

(iii) the statement STOP.

A Program Schema Π is a triple (P,I,U) where P is a finite sequence of statements and I,U are finite sets of variables- input and output respectively.

A loop-free program schema is a program schema that does not have loops- i.e. a transfer address of a test statement never references a statement which precedes the test statement in some possible sequence of statements of Π.

We will deal with loop-free program schemata, thus throughout this thesis a program schema should mean a loop-free program schema.

No transfer address references a numeral which is not a prefix of some statement, and no statement has the same prefix as another statement in the program schema.

EXAMPLE 1

$\Pi = (P,\{X,Y\},\{Z\})$

P:

$L_1 \leftarrow \varphi XY$

$t(L_1)$ 3,5

3 $Z \leftarrow \Psi L_1$

STOP

5 $L_2 \leftarrow \theta L_1$

$Z \leftarrow \Psi L_2$

STOP

$\varphi, \Psi, \theta \epsilon \Theta$ $X,Y,Z,L_1,L_2 \epsilon \Sigma$ $t \epsilon T$

A program schema represents a family of computer programs. To provide an interpretation for a program schema we choose some finite or infinite set of values (domain), then make an assignment of values from the set to each input variable and assignments of appropriate functions and predicates on the set to the operator and test names of the program schema. Given such an interpretation the schema becomes a program which can be executed by a computer.

An interpretation for the program schema Π of example 1 could be as follows:

The domain set is the set of real numbers.

The test name $t(x)$ is interpreted as the predicate $In(t)(x)$ which is true if $x \geq 0$ and is false otherwise. The interpretation of the operator names φ, Ψ, θ is as follows:

$In(\varphi XY) = +XY$

$In(\Psi X) = SQRT\ X$

$In(\theta X) = -X$

The interpreted program is

```
In(P):   L1 ← +XY
         ≥0(L1) 3,5
      3  Z  ←  SQRT L1
         STOP
      5  L2 ←  -L1
         Z  ← SQRT L2
         STOP
```

Under this interpretation the program schema computes the square root of the absolute value of X+Y.

Formally, an interpretation In of a program Π is a mapping from the set of input variables, operator names and test names of Π into a set D and the set of functions and predicates such that

(i) each variable $A \in I$ is assigned an element $In(A) \in D$.

(ii) each r-ary operator name θ is assigned an r-ary function $In(\theta): D^r \to D$.

(iii) each test name t is assigned a predicate on D, $In(t): D \to \{T,F\}$.

We shall call program schemata abstract programs or programs. An interpretation is called an actual program or a computer program.

A block is a sequence of assignment statements $S_1,\ldots,S_{n-1}$ with a STOP or a test statement S_n, and either

(i) S_1 follows a test statement

or

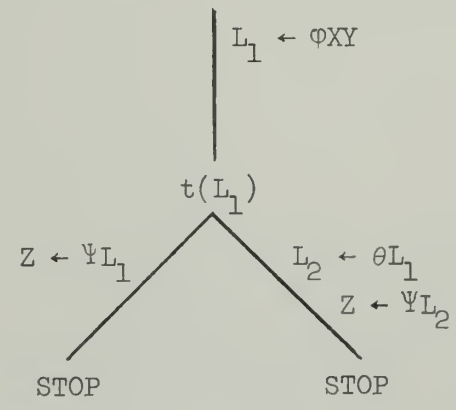
(ii) S_1 follows a STOP statement

or

(iii) S_1 is the first statement in P.

P, the sequence of statements of the program schema can be represented by a directed graph. The graph representing P will be called the graph corresponding to the program schema or simply, the graph of the program schema.

The graph corresponding to the program Π of Example 1 is



A directed graph corresponding to a program schema has the following properties:

(i) it is acyclic (i.e. it has no loops).

(ii) it has only one root.

(iii) each node can have one or two descendants, or it is a leaf.

(iv) each node can have several ancestors.

We will often represent a program schema $\Pi = (P,I,U)$ by (G_P,I,U) where G_P is the directed graph representing P.

Each path of the graph of Π corresponds to a possible sequence of statements of the program Π.

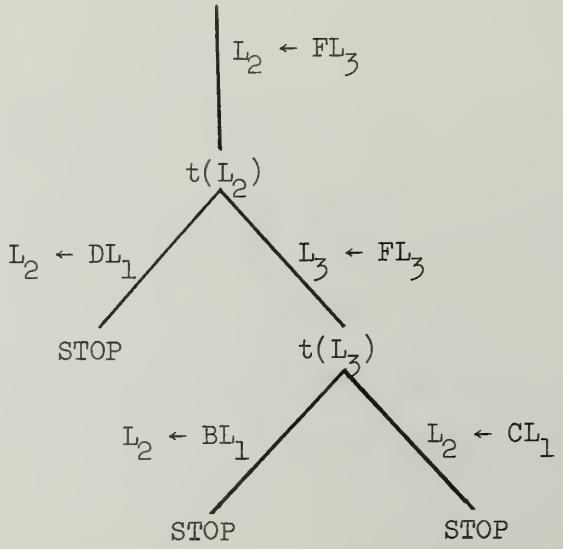
A path will be called executable if the corresponding sequence of statements can be executed under some interpretation of the program schema.

A path will be called nonexecutable if under no interpretation (i.e. assignments of input values, functions and predicates), the corresponding sequence of statements can be executed.

EXAMPLE 2

$$\Pi = (G,\{L_1,L_3\}, \{L_2\})$$

G:



The path corresponding to the sequence of statements

$L_2 \leftarrow FL_3$

$t(L_2)$

$L_3 \leftarrow FL_3$

$t(L_3)$

$L_2 \leftarrow BL_1$

STOP

is nonexecutable. Under no assignment of values to the input variables L_1 and L_3 and functions to F,B and a predicate to t this sequence of statements can be executed.

In the following chapter we shall present a theorem which characterizes executable and nonexecutable paths in terms of the test statements appearing in the paths.

3. EQUIVALENCE OF PROGRAM SCHEMATA

Two program schemata will be called equivalent iff for every interpretation the output values of the two actual programs are equal. Formally the notion of program equivalence will be defined in the following way:

Let L be the finite set of all executable paths of the program Π. For each path $\ell \epsilon L$ there is a corresponding sequence of statements $S_1^\ell, \ldots, S_m^\ell$.

If path ℓ is executed under some interpretation In, $val_{In}(\Pi)$ will be defined to be the vector of the output variables after executing $S_1^\ell, \ldots, S_m^\ell$, and will be called the value of the program schema under interpretation In.

DEFINITION

Two programs schemata Π and Π' will be called equivalent ($\Pi \equiv \Pi'$) iff for all interpretations In $val_{In}(\Pi) = val_{In}(\Pi')$.

Equivalence of program schemata will be characterized in terms of the sets of terminal expressions computed along executable paths.

We define the expression $v_k^\ell(A)$, the value of the variable A after executing S_k^ℓ as follows:

1) $v_0^\ell(A) = A$ for all $A \epsilon I$.

2) If S_k^ℓ is a test or a STOP statement $v_k^\ell(A) = v_{k-1}^\ell(A)$.

3) If S_k^ℓ is $A \leftarrow \varphi B_1 \ldots B_r$ then $v_k^\ell(A)$ is the expression

$$\varphi\, v_{k-1}^\ell(B_1) \ldots v_{k-1}^\ell(B_r)$$

and for all C, $C \neq A$, $v_k^\ell(C) = v_{k-1}^\ell(C)$.

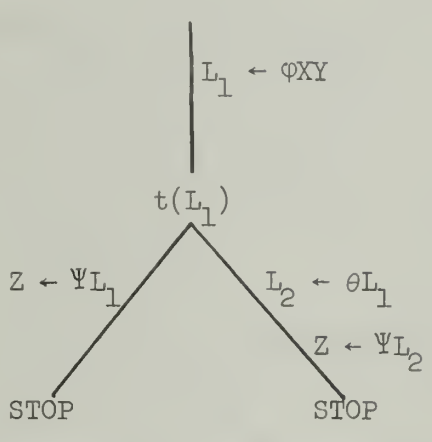
4) $v_k^\ell(A)$ is undefined otherwise.

The value of the executable path ℓ, denoted by $v^{\ell}(\Pi)$ is

$$\{v_m^{\ell}(A),\ A \epsilon U\}$$

EXAMPLE 3

If Π is of Example 1, $\Pi = (G_P, \{X,Y\}, \{Z\})$



and ℓ_1 and ℓ_2 are the left and right paths respectively, then

$$v^{\ell_1}(\Pi) = v_4^{\ell_1}(Z) = \Psi\varphi XY$$

$$v^{\ell_2}(\Pi) = v_5^{\ell_2}(Z) = \Psi\theta\varphi XY$$

A program Π is said to be proper if $\forall\ \ell$, ℓ is an executable path, whenever a variable B is referenced by the kth statement of ℓ (S_k^{ℓ}) then $v_{k-1}^{\ell}(B)$ is defined.

We will deal with proper programs only, thus throughout this thesis, a program schema or a program should mean a proper loop-free program schema.

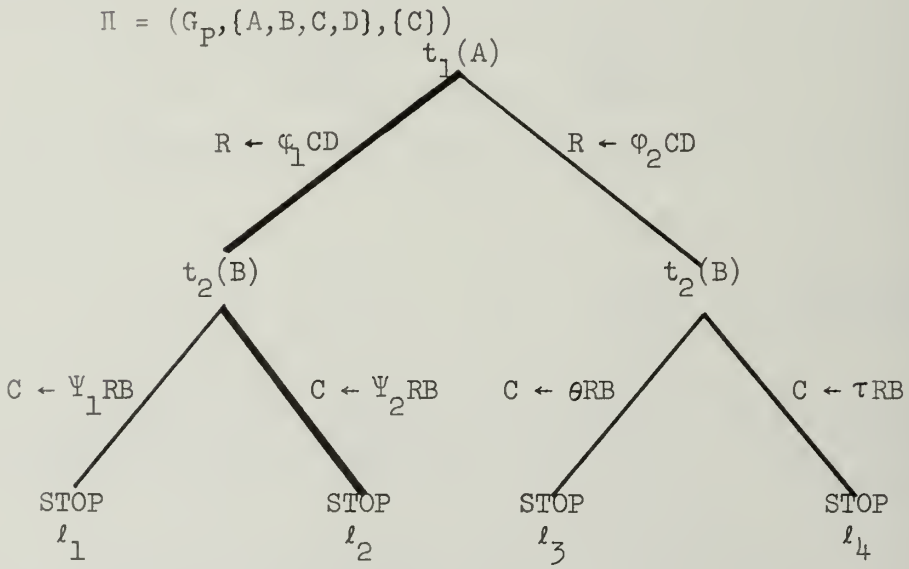
DEFINITION

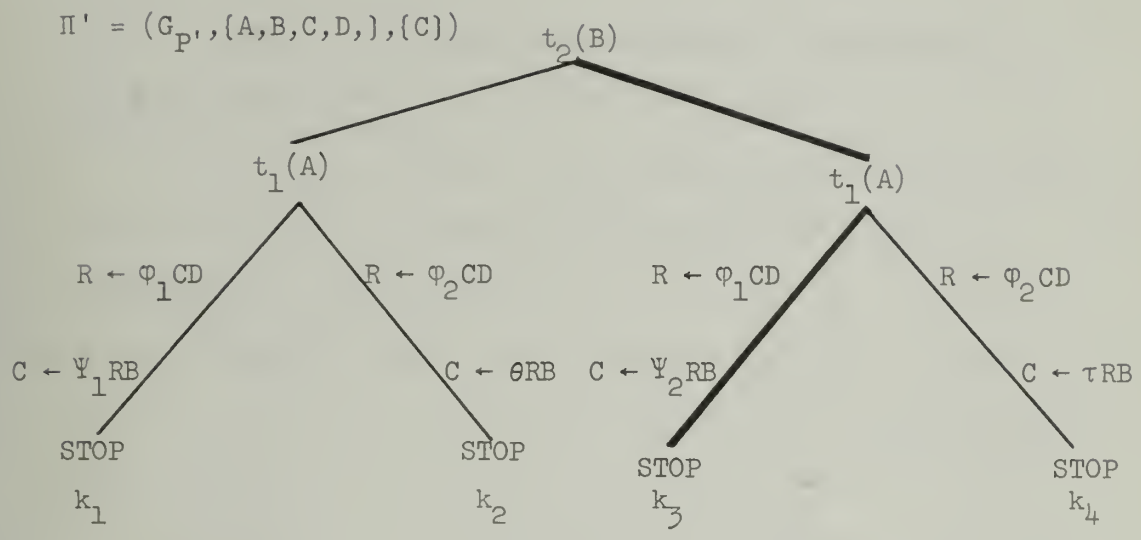
Let Π and Π' be two programs. Let ℓ and k be two executable paths of Π and Π' respectively. ℓ and k are said to be consistent iff $\forall$t such that

(i) t is in the sequences of statements that correspond to both ℓ and k, so that in path ℓ there is a statement S_i^{ℓ} of the form $t^{\ell}(C_1,\ldots,C_r)n_1,n_2$, and in path k there is a statement S_j^k of the form $t^k(D_1,\ldots,D_r)m_1,m_2$ and

(ii) $v_i^{\ell}(C_q) = v_j^k(D_q)$ for all q, $1 \leq q \leq r$, that is the values of the variables referenced by the statements are identical, then the statement prefixed by n_1 is included in path ℓ iff the statement prefixed by m_1 is included in path k, and the statement prefixed by n_2 is included in path ℓ iff the statement prefixed by m_2 is included in path k.

EXAMPLE 5





The marked paths are consistent.

The following theorem gives a simple characterization of executable and nonexecutable paths in terms of the test statements occuring in the paths.

THEOREM 1

A path ℓ of a program Π is executable iff there are no two test statements in ℓ that reference variables that have the same values and give different truth values.

PROOF:

1) If there are two test statements that reference variables that have the same values and give defferent truth values, this path can not be executed under any interpretation, and therefore is nonexecutable. Thus if the path is executable there are no tests with the above property.

2) Assume there are no two tests with the above property. Then path ℓ will be executed under the following interpretation In:

The domain set of In is the set of strings of variables and operator names.

If A is a variable, $In(A) = A$.

If φ is a function name, $\alpha_1,\ldots,\alpha_n$ are strings, then $In(\varphi\alpha_1\ldots\alpha_n)$ is $\varphi\alpha_1\ldots\alpha_n$, the concatenation of φ with $\alpha_1,\ldots,\alpha_n$.

If t^ℓ is a test name in path ℓ,

1) $In(t^\ell)(\alpha_1,\ldots,\alpha_n) = T$ for all $(\alpha_1,\ldots,\alpha_n)$ such that $\alpha_i = v_j^\ell(X_i)$, $1 \leq i \leq n$ and $t^\ell(X_1,\ldots,X_n) = T$ for some appearance S_j^ℓ of t^ℓ in path ℓ.

2) $In(t^\ell)(\alpha_1,\ldots,\alpha_n) = F$ for all $(\alpha_1,\ldots,\alpha_n)$ such that $\alpha_i = v_j^\ell(X_i)$ $1 \leq i \leq n$ and $t^\ell(X_1,\ldots,X_n) = F$ for some appearance S_j^ℓ of t^ℓ in path ℓ.

3) $In(t^\ell)(\alpha_1,\ldots,\alpha_n)$ is arbitrary for all other n-tuples $(\alpha_1,\ldots,\alpha_n)$.

Because no two tests in ℓ reference variables that have the same values and give different truth values-the definition of the tests is unique, and path ℓ is executed under the interpretation In. Therefore ℓ is executable.

THEOREM 2

$\Pi \equiv \Pi'$ iff for all consistent pairs of paths ℓ,k $v^\ell(\Pi) = v^k(\Pi')$.

Theorem 2 states that two programs are equivalent iff the sets of expressions computed along consistent pairs of paths are identical. Thus deciding if two programs are equivalent can be done by checking that terminal expressions are identical for consistent paths.

Two programs Π and Π' always have consistent pairs of paths ℓ,k. For each path in one program there is always a consistent path in the other.

The proof is based on Luckham, Park and Paterson(8).

PROOF:

1) Let $\Pi \equiv \Pi'$, and ℓ and k are consistent. $\Pi \equiv \Pi'$ therefore by definition for all interpretations In $val_{In}(\Pi) = val_{In}(\Pi')$. Take In as the following interpretation:

The domain set of In, the interpretation of inputs and functions, are the same as in Theorem 1. The test names are interpreted in the following way:

If t^j is a test name in path $j = \ell,k$

1) $In(t^j)(\alpha_1,\ldots,\alpha_n) = T$ for all $(\alpha_1,\ldots,\alpha_n)$ such that $\alpha_i = v_m^j(X_i)$ $1 \leq i \leq n$ and $t^j(X_1,\ldots,X_n) = T$ for some appearance S_m^j of t^j in paths $j=\ell,k$.

2) $In(t^j)(\alpha_1,\ldots,\alpha_n) = F$ for all $(\alpha_1,\ldots,\alpha_n)$ such that $\alpha_i = v_m^j(X_i)$, $1 \leq i \leq n$ and $t^j(X_1,\ldots,X_n) = F$ for some appearance S_m^j of t^j in paths $j=\ell,k$.

3) In $(t^j)(\alpha_1,\ldots,\alpha_n)$ is arbitrary for all other n - tuples $(\alpha_1,\ldots,\alpha_n)$.

ℓ,k are consistent, therefore the definition of the tests is unique. $val_{In}(\Pi)$ was defined to be the vector of values of the output variables after executing the program under interpretation In. Under the interpretation In above path ℓ in Π is executed and $val_{In}(\Pi) = v^{\ell}(\Pi)$. Also under this interpretation path k in Π' is executed and $val_{In}(\Pi') = v^k(\Pi')$. Therefore $v^{\ell}(\Pi) = v^k(\Pi')$.

2) Let $v^{\ell}(\Pi) = v^k(\Pi')$ for all consistent pairs of paths ℓ,k. Let In be any interpretation. We have to show that $val_{In}(\Pi) = val_{In}(\Pi')$.

Let ℓ,k be the paths executed under the interpretation In in Π and Π' respectively. Then ℓ,k are consistent. Then $v^{\ell}(\Pi) = v^{k}(\Pi')$. But if $v^{\ell}(\Pi) = v^{k}(\Pi')$ then $val_{In}(\Pi) = val_{In}(\Pi')$ because if the expressions for the output variables are equal, then if we substitute the functions and inputs we will get equal values. (We used here the fact that the Polish notation representation of expressions is nonambiguous).

EXAMPLE 6

a) Π and Π' of Example 5 are equivalent. The consistent pairs of paths are ℓ_1 and k_1, ℓ_2 and k_3, ℓ_3 and k_2, ℓ_4 and k_4.

$v^{\ell_1}(\Pi) = \Psi_1\varphi_1 CDB$ $\qquad$ $v^{k_1}(\Pi') = \Psi_1\varphi_1 CDB$

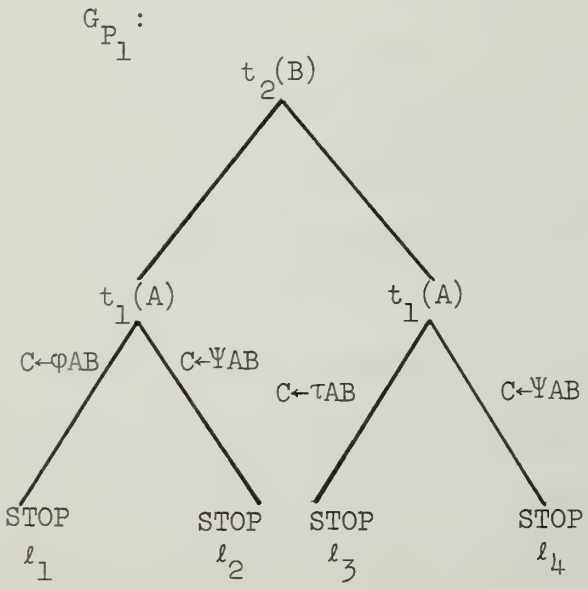
$v^{\ell_2}(\Pi) = \Psi_2\varphi_1 CDB$ $\qquad$ $v^{k_3}(\Pi') = \Psi_2\varphi_1 CDB$

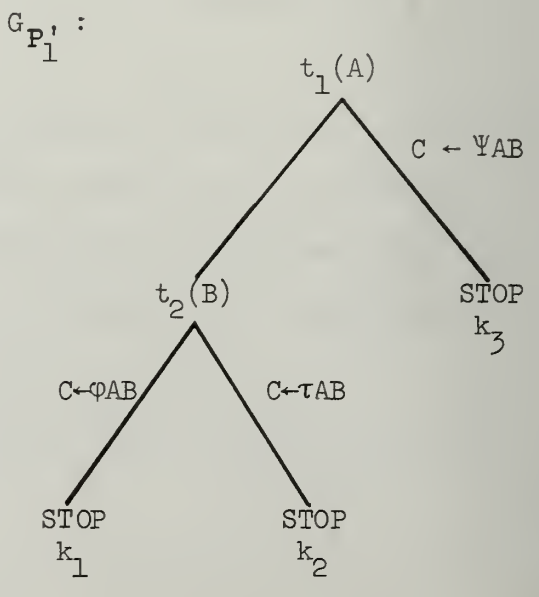
$v^{\ell_3}(\Pi) = \theta\varphi_2 CDB$ $\qquad$ $v^{k_2}(\Pi') = \theta\varphi_2 CDB$

$v^{\ell_4}(\Pi) = \tau\varphi_2 CDB$ $\qquad$ $v^{k_4}(\Pi') = \tau\varphi_2 CDB$

Therefore for all consistent pairs of paths the expressions computed are identical. Thus $\Pi \equiv \Pi'$.

b) $\Pi_1 = (G_{P_1}, \{A,B\},\{C\})$ $\qquad$ $\Pi_1' = (G_{P_1'}, \{A,B\}, \{C\})$





In this example the consistent pairs of paths are (ℓ_1, k_1), (ℓ_2, k_3), (ℓ_3, k_2), (ℓ_4, k_3).

$\Pi_1 \equiv \Pi_1'$.

4. TRANSFORMATIONS ON PROGRAMS

Let Π be a program, and let ℓ be a path of the directed graph with the corresponding sequence of statements $S_1^\ell,\ldots,S_n^\ell$.

Suppose S_i^ℓ defines the variable A.

1) If $A \notin U$, and S_j^ℓ is the last statement to reference this instance of A (i.e. for no $k > j$ S_k^ℓ references A, unless A is defined by some S_t^ℓ $j < t < k$) then the scope of S_i^ℓ in path ℓ is the sequence of statements $S_{i+1}^\ell,\ldots,S_j^\ell$.

2) $A \in U$, and for no $j > i$ S_j^ℓ defines A, then the scope of S_i^ℓ in path ℓ is $S_{i+1}^\ell,\ldots,S_n^\ell$ and the set U.

3) If no S_j^ℓ, $j > i$ references this instance of A, and A is not an output variable, then the scope of S_i^ℓ is null, and S_i^ℓ is said to be useless in path ℓ.

Any program can be represented by a set of labeled directed acyclic graphs (dags). For each executable path ℓ of the program we construct a dag $D_\ell(\Pi)$ as follows:

i) For all A, $A \in I$, we create a leaf labeled by A.

ii) If $S_1^\ell,\ldots,S_n^\ell$ is the corresponding sequence of statements, for each $j = 1,\ldots,n$ we check if S_j^ℓ is an assignment statement. If yes - we create a node associated with it. If S_j^ℓ is of the form

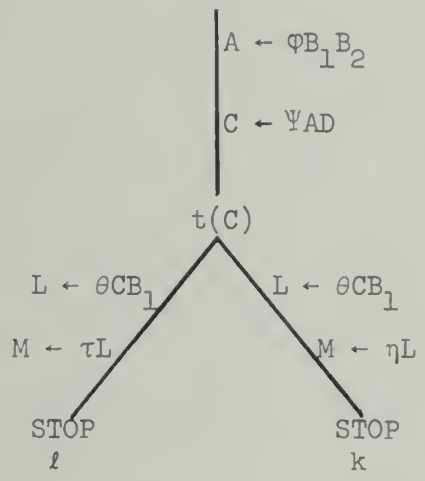
$$A \leftarrow \theta B_1 \ldots B_r$$

and $n_1,\ldots,n_r$ are the nodes associated with the most recent definitions of $B_1,\ldots,B_r$ respectively (or if some $B_k \in I$, n_k is the leaf with that label), then the node associated with S_j has a label θ and direct descendants $n_1,\ldots,n_r$.

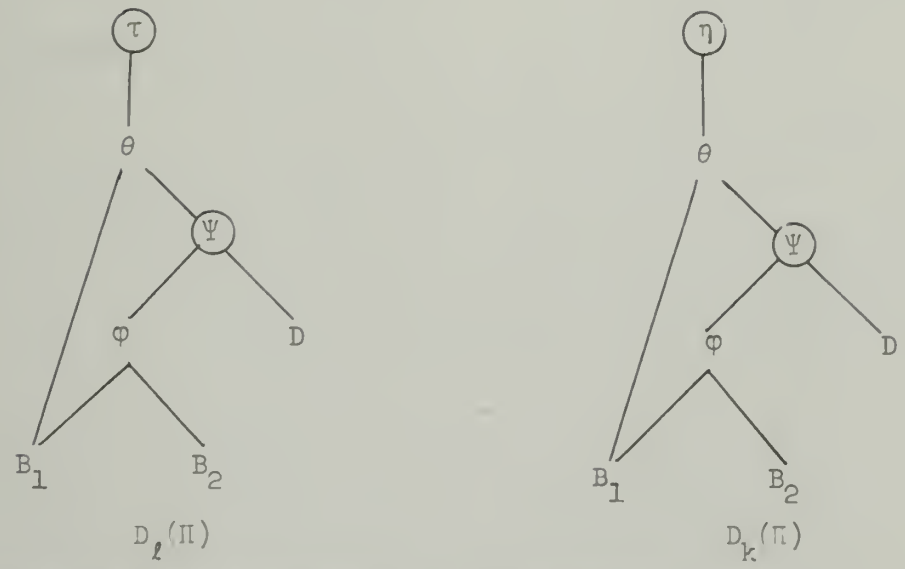
iii) We distinguish by circling those nodes associated with the last definitions of the output variables or the nodes associated with definitions of variables referenced by test statements.

EXAMPLE 7

$\Pi = (G,\{B_1,B_2,D\},\{M\})$



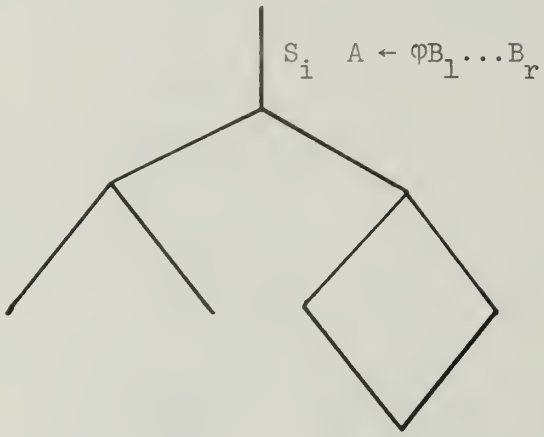
$D(\Pi)$, the set of dags of Π, is



ℓ and k are the left and right paths, respectively.

A transformation on a program is a mapping to the set of programs which preserves program equivalence. We will define a set of transformations that operate on loop-free programs.

T1 Removal of Useless Assignment Statements

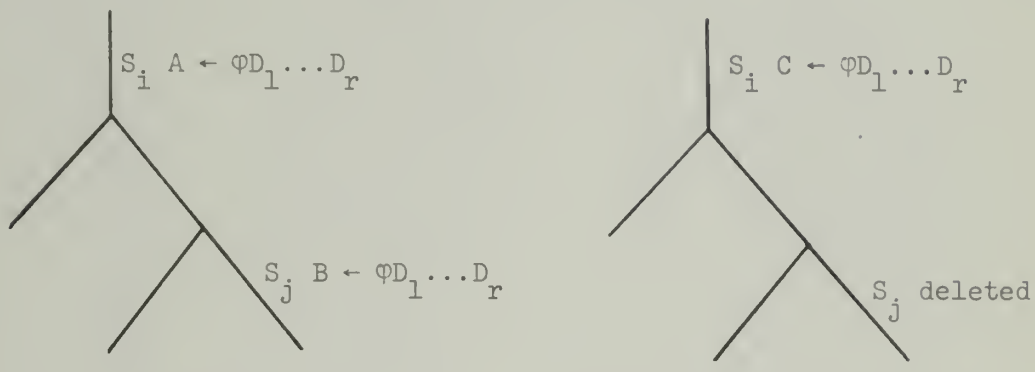


If for all paths ℓ, such that S_i is in the corresponding sequence of statements of ℓ, S_i is useless in ℓ, then S_i can be removed from the program. Also, if $A \epsilon I$, $A \notin U$, A is not referenced in Π, A can be removed from I.

If S_i is removed, all references to S_i (addresses of test statements) are changed to reference S_{i+1}.

T1 operates on all the dages D_{ℓ_j} corresponding to paths ℓ_j which include the useless statement S_i. T1 deletes a node which is not distinguished and has no ancestors.

T2 Removal of Redundant Assignment Statements



If S_i^ℓ is A ← φ$D_1 \ldots D_r$

S_j^ℓ is B ← φ$D_1 \ldots D_r$

S_i^ℓ and S_j^ℓ are in the sequence of statements that correspond to path ℓ, and there is no path that includes S_j^ℓ and not S_i^ℓ, and also for each path that includes both S_i^ℓ and S_j^ℓ, $D_1, \ldots, D_r$ are not defined by any S_k^ℓ $i \leq k < j$, then S_i^ℓ is replaced by

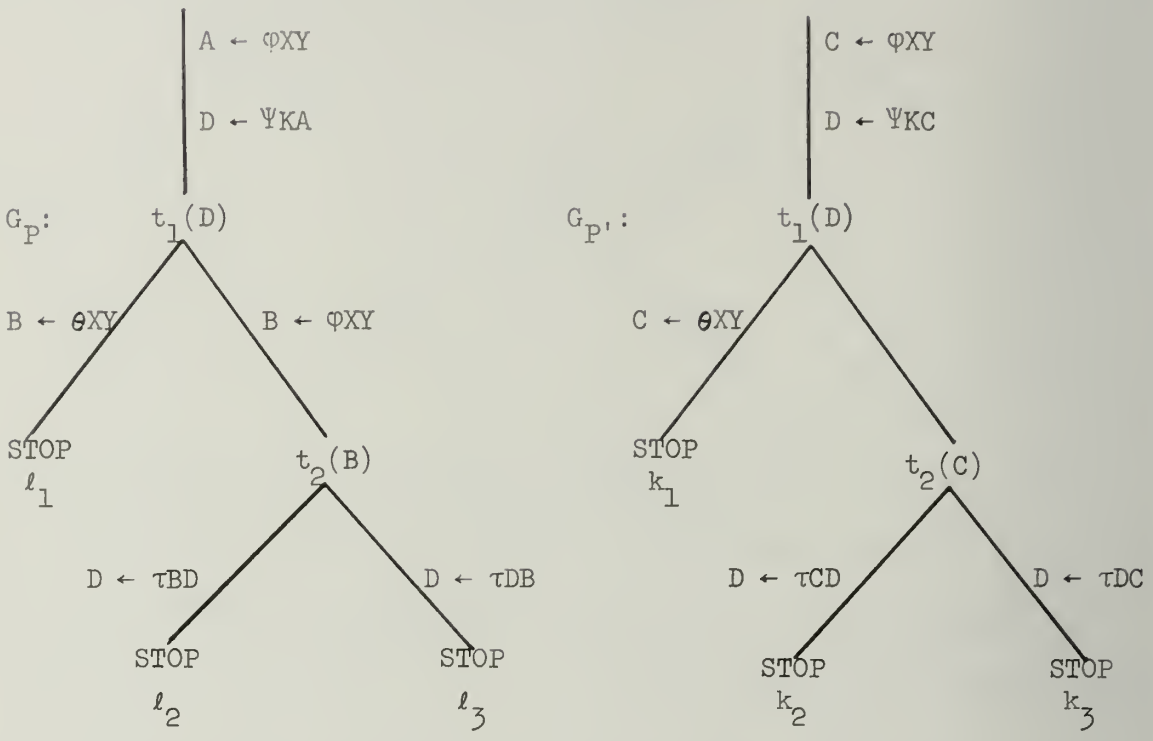
$$C \leftarrow \varphi D_1 \ldots D_r,$$

S_j^ℓ is deleted, and all references to A and B in the scopes of S_i^ℓ and S_j^ℓ in all the paths that include S_i^ℓ are changed to references to C. All references to S_j^ℓ are replaced by references to S_{j+1}^ℓ.

T2 operates on all the dags $D_{\ell_k}(\Pi)$ corresponding to paths ℓ_k which include both S_i^ℓ and S_j^ℓ. All the other dags remain unchanged. T2 corresponds to the merging of two nodes with identical direct descendants.

EXAMPLE 8

$\Pi = (P, \{K,X,Y\}, \{B,D\})$ $\qquad\qquad$ $\Pi' = (P', \{K,X,Y\}, \{C,D\})$



(ℓ_1,k_1), (ℓ_2,k_2), (ℓ_3,k_3) are the consistent pairs of paths.

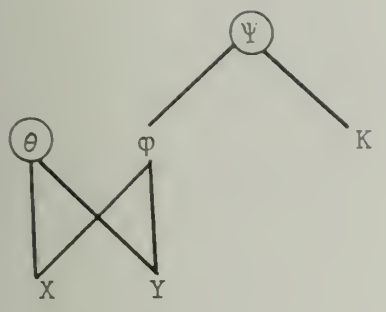
$v^{\ell_1}(\Pi) = \{\theta XY, \Psi K\varphi XY\} = v^{k_1}(\Pi')$

$v^{\ell_2}(\Pi) = \{\varphi XY, \tau\varphi XY\Psi K\varphi XY\} = v^{k_2}(\Pi')$

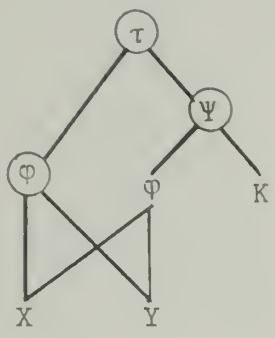
$v^{\ell_3}(\Pi) = \{\varphi XY, \tau\Psi K\varphi XY\varphi XY\} = v^{k_3}(\Pi')$

T2 transforms Π to Π'.

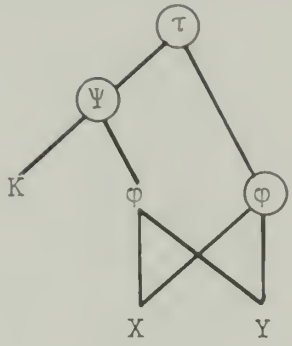
The sets of dags for Π and Π' are shown on the next page.



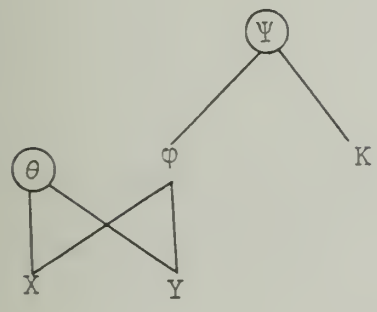
$D_{\ell_1}(\Pi)$



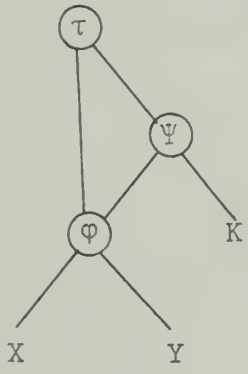
$D_{\ell_2}(\Pi)$



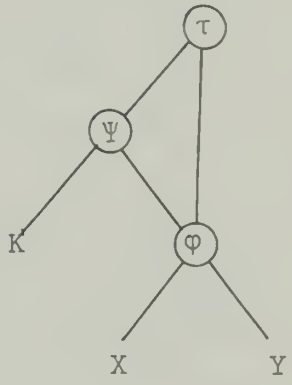
$D_{\ell_3}(\Pi)$



$D_{k_1}(\Pi')$



$D_{k_2}(\Pi')$



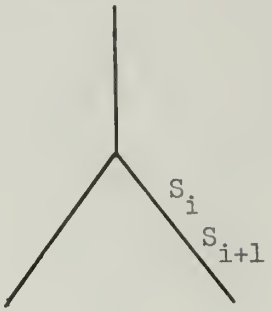
$D_{k_3}(\Pi')$

T3 <u>Renaming</u>

$S_i \quad A \leftarrow \Phi B_1 \ldots B_r$

We replace S_i by S_i' - $C \leftarrow \Phi B_1 \ldots B_r$, and all references to A in the scopes of S_i in all the paths that include S_i are replaced by references to C. Variable names do not appear in a dag, therefore the dags are not affected.

T4 Flipping of Assignment Statements



S_i (k) $A \leftarrow \Phi B_1 \ldots B_r$

S_{i+1} (k+1) $C \leftarrow \Psi D_1 \ldots D_q$

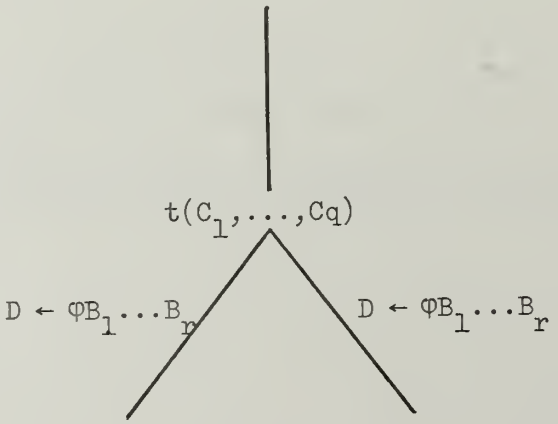
$A \neq \{C, D_1, \ldots, D_q\}$

$C \neq \{B_1, \ldots, B_r\}$

and there is no path that includes S_{i+1} and not S_i. Then S_i and S_{i+1} may be interchanged. If numerals precede the statements, they are interchanged accordingly. T4 does not affect the dags.

T5 Merging of Identical Assignment Statements

(a)



If the statement $t(C_1,\ldots,C_q)\ k_1,k_2$ is in paths ℓ and ℓ' i.e.

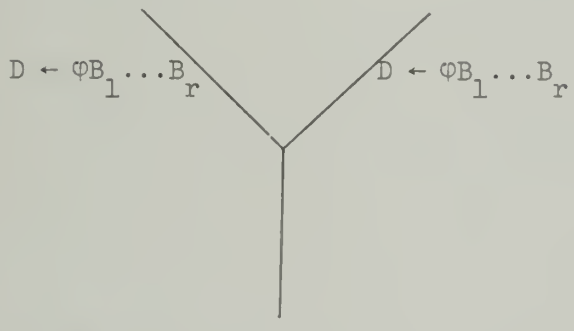
$$S_i^{\ell} = S_i^{\ell'} = t(C_1,\ldots,C_q)\ k_1,k_2$$

and $$S_{i+1}^{\ell} = S_{i+1}^{\ell'} = D \leftarrow \Phi B_1 \ldots B_r$$

and $$D \neq \{C_1,\ldots,C_q\}$$

and there is no path that includes $S_{i+1}^{\ell}(S_{i+1}^{\ell'})$ and does not include $t(C_1,\ldots,C_q)$, then the statement $D \leftarrow \Phi B_1 \ldots B_r$ is moved before $t(C_1,\ldots,C_q)$ and S_{i+1}^{ℓ}, $S_{i+1}^{\ell'}$ are deleted. The prefix of the statement $D \leftarrow \Phi B_1 \ldots D_r$ is that of $t(C_1,\ldots,C_q)$. The prefix of $t(C_1,\ldots,C_q)$ is deleted. k_1,k_2 in the test statement are replaced by references to S_{i+2}^{ℓ}, $S_{i+2}^{\ell'}$.

(b)



If $S_i^{\ell} = S_i^{\ell'} = D \leftarrow \Phi B_1 \ldots B_r$

and S_i^{ℓ}, $S_i^{\ell'}$ are the last assignment statements before ℓ and ℓ' are merged, then S_i^{ℓ} and $S_i^{\ell'}$ can be merged. All references to S_i^{ℓ}, $S_i^{\ell'}$ are replaced by references to the merged statement. T5 does not affect the dags.

EXAMPLE 9

Π = (P,{A,B}, {L})

P = t_1(A) 2,5

2 N ← ΦAB

L ← ΨNN

t(B) 7,7

5 N ← θAB

L ← ΨNN

7 L ← τLL

STOP

Π' = (P',{A,B}, {L})

P' = t_1(A) 2,5

2 N ← ΦAB

t(B) 7,7

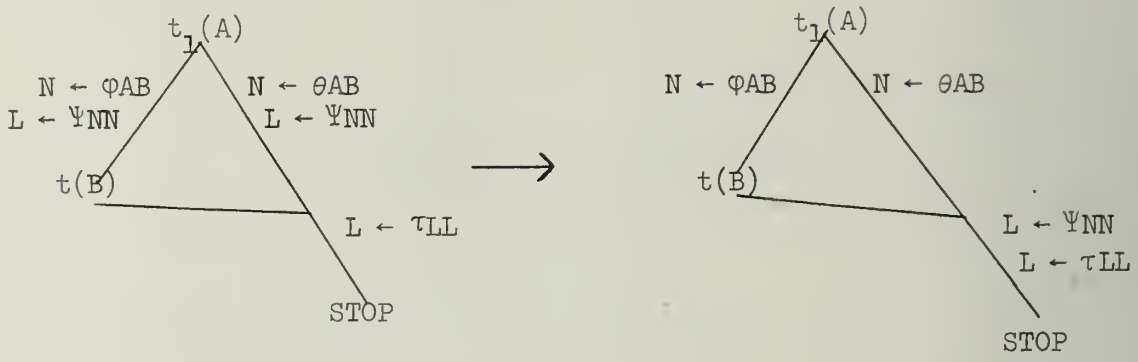
5 N ← θAB

7 L ← ΨNN

L ← τLL

STOP

T5 transforms Π to Π'.



Π Π'

T6 Removal of Nonexecutable Paths

(a) If S_i^ℓ = $t(A_1,\ldots,A_r)$ k_1,k_2

S_j^ℓ = $t(C_1,\ldots,C_r)$ k_3,k_4 $\quad j > i$

(*) and there is no path ℓ that includes S_j^ℓ and not S_i^ℓ

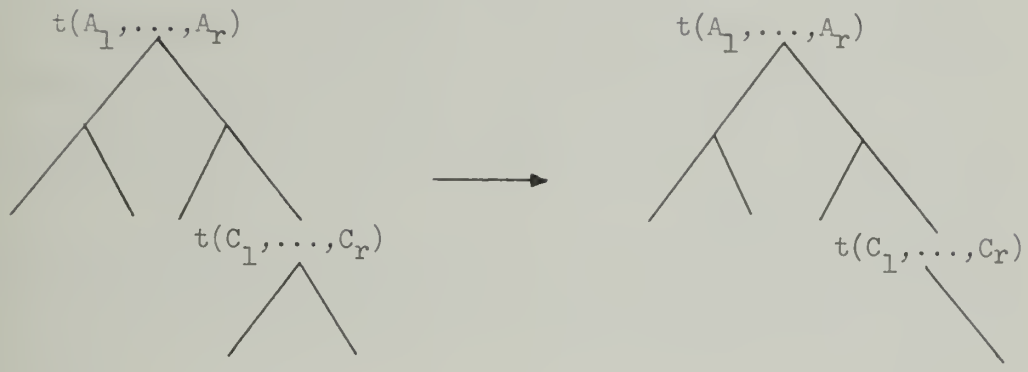
(**) and $\forall$ k, $1 \leq k \leq r$

$v_i^\ell(A_k) = v_j^\ell(C_k)$

then

(i) if S_j^ℓ is on the right branch of S_i^ℓ, S_j^ℓ is changed to $t(C_1,\ldots,C_r)k_4,k_4$.

(ii)if S_j^ℓ is on the left branch of S_i^ℓ, S_j^ℓ is changed to $t(C_1,\ldots,C_r)k_3,k_3$.



(b) If $S_i^\ell = t(A_1,\ldots A_r)k_1,k_2$, and

(i) $S_{k_1}^\ell = t(C_1,\ldots,C_r)k_3,k_4$

and (*) and (**) hold for S_i^ℓ and $S_{k_1}^\ell$ then we change S_i^ℓ to $t(A_1,\ldots,A_r)k_3,k_2$ and $S_{k_1}^\ell$ is deleted.

(ii) $S_{k_2}^\ell = t(C_1,\ldots,C_r)k_3,k_4$

and (*) and (**) hold for S_i^ℓ and $S_{k_2}^\ell$, we change S_i^ℓ to $t(A_1,\ldots,A_r)k_1,k_4$ and $S_{k_2}^\ell$ is deleted.

Since T6 gets rid of tests, it uncircles nodes in the dags that are associated only with variables referenced by the tests removed.

T7 Removal of Unconditional Useless Test Statements

If S_i is of the form $t(C_1,\ldots,C_n)k_1,k_1$ and $k_1 = i+1$, then S_i is removed. All references to S_i are changed to references to S_{i+1}.

Since T7 removes tests, it uncircles nodes in the dags that are associated only with variables referenced by the tests removed.

T8 Flipping of Tests

If $S_i^{\ell} = t_1(A_1,\ldots,A_r)\ k_1,k_2$ and S_i is on both paths ℓ and ℓ', $S_i^{\ell} = S_{i'}^{\ell'}$,

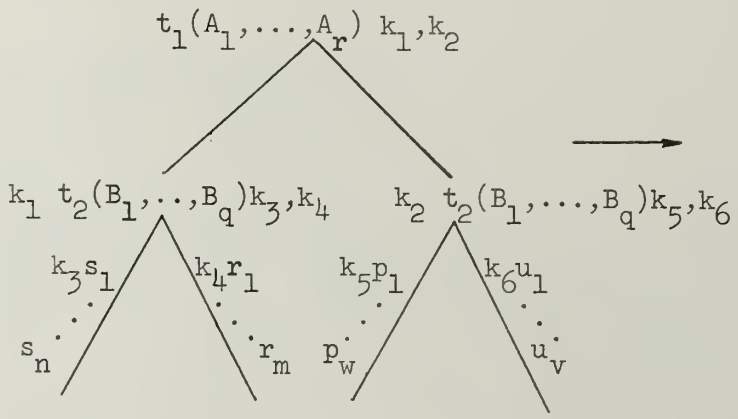
and $S_{k_1}^{\ell} = t_2(B_1,\ldots,B_q)\ k_3,k_4$

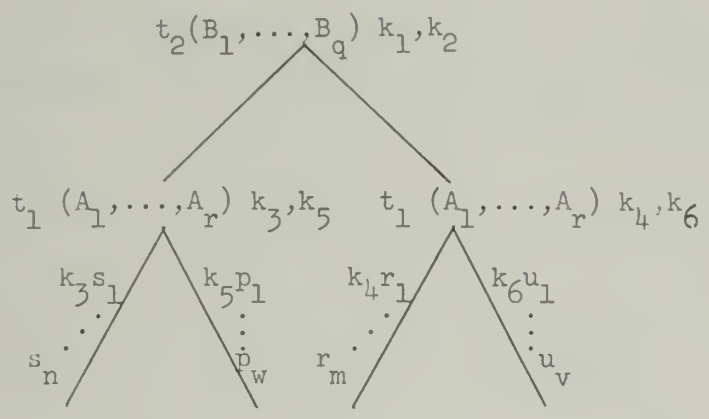
and $S_{k_2}^{\ell'} = t_2(B_1,\ldots,B_q)\ k_5,k_6$

then t_1 and t_2 can be flipped so that $S_i^{\ell} = S_{i'}^{\ell'} = t_2(B_1,\ldots,B_q)\ k_1,k_2$.

$S_{k_1}^{\ell} = t_1(A_1,\ldots,A_r)\ k_3,k_5$

$S_{k_2}^{\ell'} = t_1(A_1,\ldots,A_r)\ k_4,k_6$.





The sequences of statements for the same combinations of tests are not changed, therefore the transformation does not affect the set of dags.

T9 Removal of Unreferenced Blocks

If the first statement of a block is not referenced (i.e. either the numeral is not a transfer address of any of the statements of P, or the first statement of the block is not prefixed by a numeral and the block is not the first in the program) then the whole block is removed.

If the first statement of a block is not referenced, there is no executable sequence of statements that contains the block, thus T9 does not affect the dags.

An example of the use of T9 is removing a useless STOP statement:

If S_i is STOP

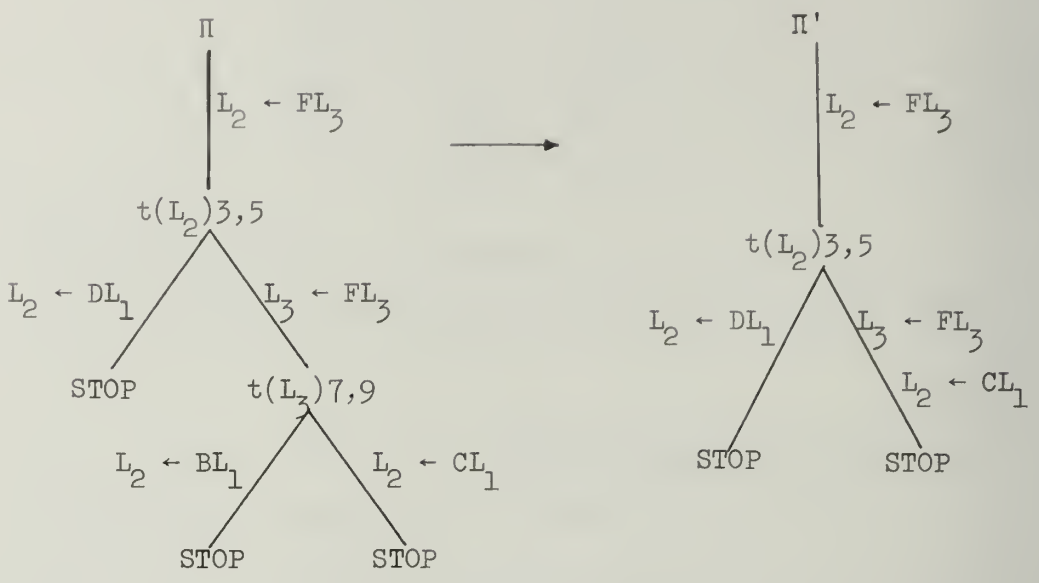
and

S_{i+1} is STOP

and there is no path that includes S_{i+1} and does not include S_i, then T9 can be used to remove S_{i+1}.

EXAMPLE 10

Π is of Example 2. We would like to remove the nonexecutable path.



Π = (P,{L_1,L_3}, {L_2})

P =

$L_2 \leftarrow FL_3$		$L_2 \leftarrow FL_3$		$L_2 \leftarrow FL_3$
$t(L_2)$ 3,5		$t(L_2)$ 3,5		$t(L_2)$ 3,5
3 $L_2 \leftarrow DL_1$		3 $L_2 \leftarrow DL_1$		3 $L_2 \leftarrow DL_1$
STOP	T6 →	STOP	T9 →	STOP	T7 →
5 $L_3 \leftarrow FL_3$		5 $L_3 \leftarrow FL_3$		5 $L_3 \leftarrow FL_3$
$t(L_3)$ 7,9		$t(L_3)$ 9,9		$t(L_3)$ 9,9
7 $L_2 \leftarrow BL_1$		7 $L_2 \leftarrow BL_1$		9 $L_2 \leftarrow CL_1$
STOP		STOP		STOP
9 $L_2 \leftarrow CL_1$		9 $L_2 \leftarrow CL_1$		
STOP		STOP		
Π				

$L_2 \leftarrow FL_3$

$t(L_2)$ 3,5

3 $L_2 \leftarrow DL_1$

STOP

5 $L_3 \leftarrow FL_3$

9 $L_2 \leftarrow CL_1$

STOP

Π'

T10 Flipping of Blocks

Any two blocks (except the first) can be flipped. The graph of the program is not changed, and the dags are not affected.

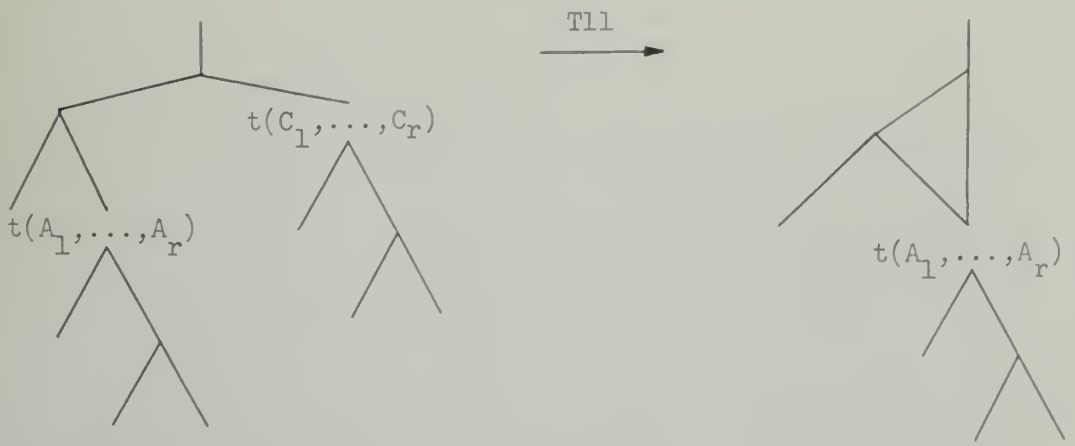
T11 Merging of Identical Subgraphs

a) if $S_i^{\ell} = t(A_1,\ldots,A_r)\ k_1,k_2$

$S_j^{\ell'} = t(C_1,\ldots,C_r)\ k_3,k_4$

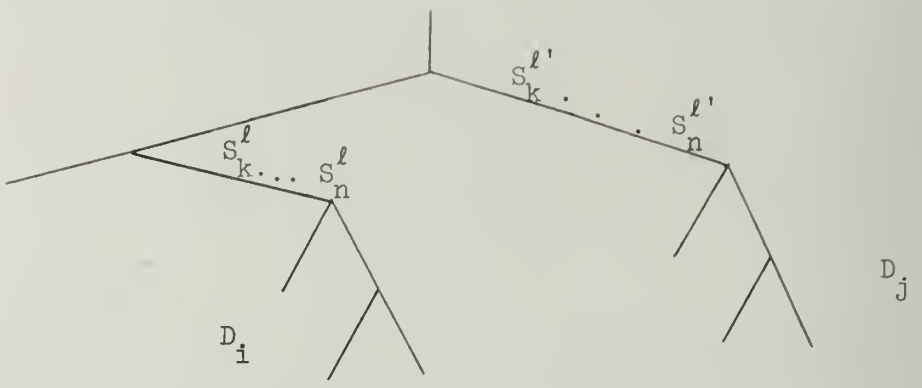
and $\forall k,\ 1 \leq k \leq r \quad v_i^{\ell}(A_k) = v_j^{\ell'}(C_k)$

and S_i^{ℓ}, $S_j^{\ell'}$ are the roots of identical subgraphs D_i, D_j, respectively, then D_i and D_j are merged, $S_j^{\ell'}$ is deleted, and all references to statements in D_j are changed to references to statements in D_i.

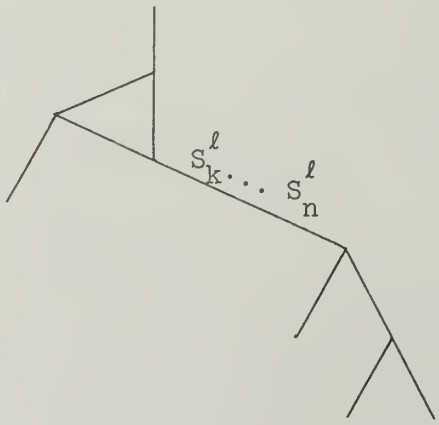


The executable paths are not affected by this transformation therefore the dags are not affected.

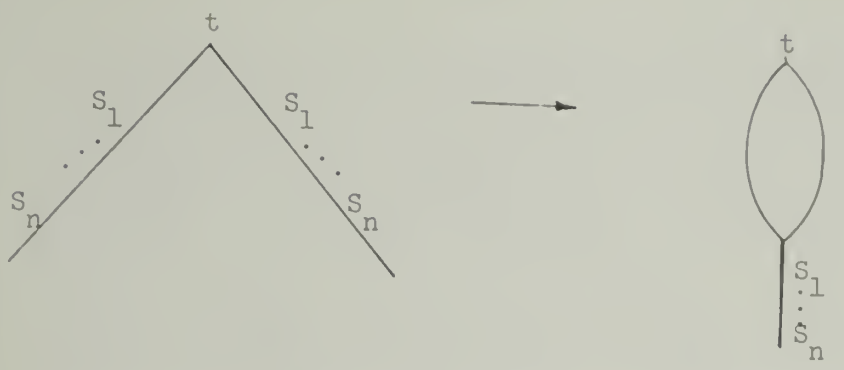
b)



Also, if the subgraphs D_i and D_j are identical, and also the sequences of assignment statements $S_k^{\ell},\ldots,S_n^{\ell}$ and $S_k^{\ell'},\ldots,S_n^{\ell'}$ preceding D_i and D_j respectively, are identical, then a test statement is added to the program and the sequences of assignment statements together with the subgraphs are merged.



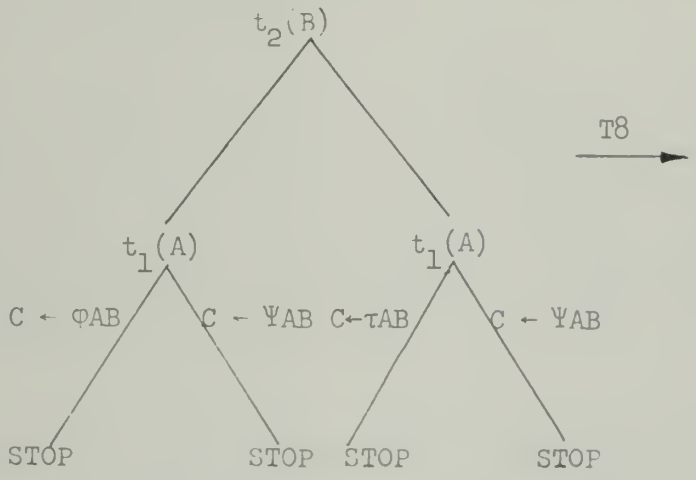
A trivial case of T11(b) which will be used later is

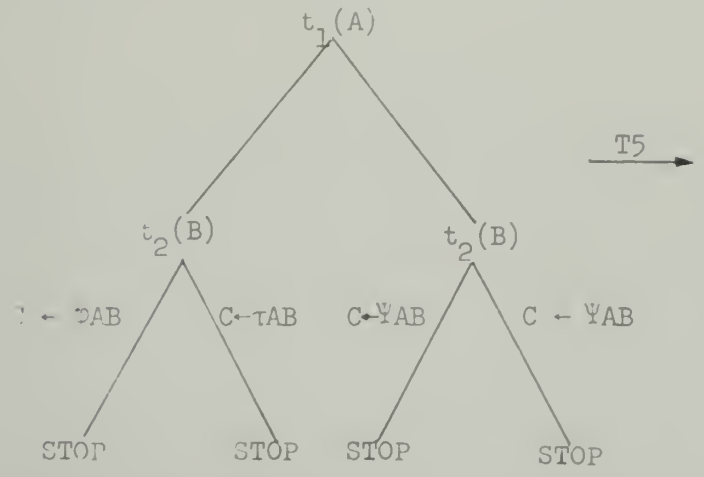


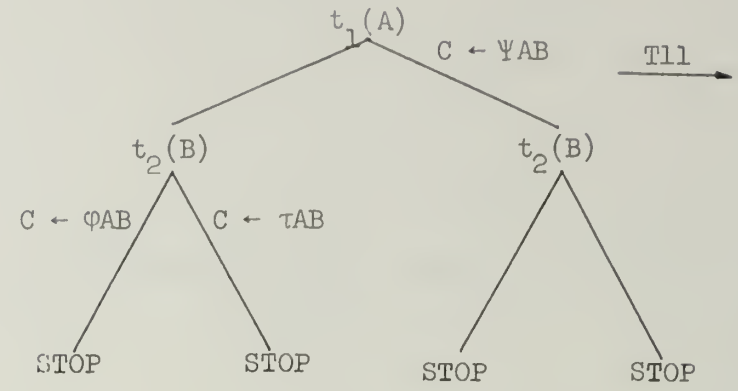
Here D_i and D_j are empty, and the identical sequences of statements are merged.

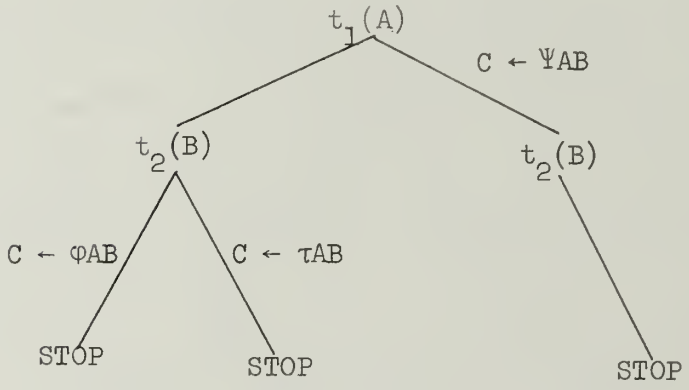
EXAMPLE 11

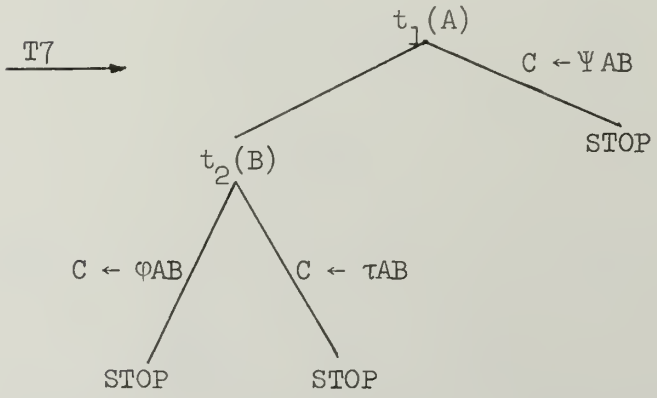
Let Π and Π' be the programs of Example 6(b). We would like to show that Π can be transformed to Π' by applying T5, T7, T8, T11.






$t_1(A)$
$C \leftarrow \Psi AB$
T11
$t_2(B)$
$t_2(B)$
$C \leftarrow \Phi AB$
$C \leftarrow \tau AB$
STOP
STOP
STOP
STOP


$t_1(A)$
$C \leftarrow \Psi AB$
$t_2(B)$
$t_2(B)$
$C \leftarrow \Phi AB$
$C \leftarrow \tau AB$
STOP
STOP
STOP


T7
$t_1(A)$
$C \leftarrow \Psi AB$
STOP
$t_2(B)$
$C \leftarrow \Phi AB$
$C \leftarrow \tau AB$
STOP
STOP

All the transformations presented above are equivalence preserving. This is clear from the definitions.

We denote $\Pi \xrightarrow[i]{} \Pi'$ if transformation Ti transforms Π to Π'. $i = 1,2,3,\ldots,11$.

We define $\xrightarrow[i]{R}$ to be the reverse of $\xrightarrow[i]{}$.

We say $\Pi \xrightarrow[S]{*} \Pi'$ where $S \subseteq \{1,2,\ldots,11\}$ if there is a sequence of programs $\Pi_1,\ldots,\Pi_m$ $\Pi_1 = \Pi, \Pi_m = \Pi'$ and for all i $\Pi_i \xrightarrow[j]{} \Pi_{i+1}$ or $\Pi_i \xrightarrow[j]{R} \Pi_{i+1}$ $j \in S$.

DEFINITION

A set of transformations J is defined to be complete iff

$$\Pi \equiv \Pi' \implies \Pi \xrightarrow[J]{*} \Pi'$$

THEOREM 3

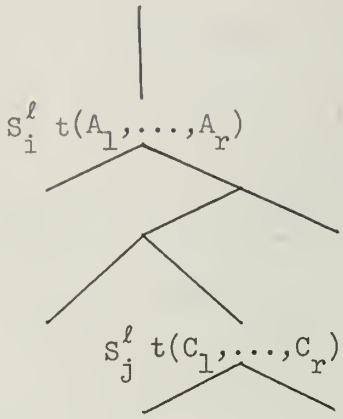
If the graph corresponding to a program Π has nonexecutable paths, then Π can be transformed to an equivalent program Π' in which each path is executable, by applying a sequence of transformations from the set S = {T6, T11}.

PROOF:

If path ℓ is nonexecutable, by Theorem 1 there must be at least two tests in path ℓ that check the same values and give different truth values.

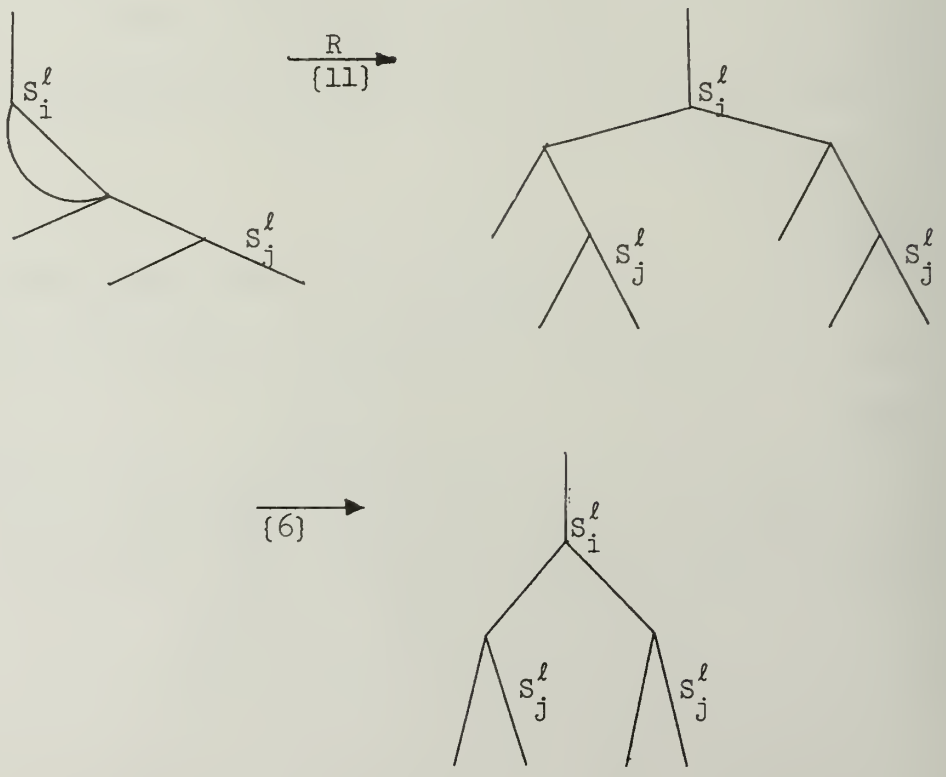
Let S_i^{ℓ} be $t(A_1,\ldots,A_r)$ and S_j^{ℓ} be $t(C_1,\ldots,C_r)$ $j > i$ and $v_i^{\ell}(A_k) = v_j^{\ell}(C_k)$ $1 \leq k \leq r$, and $t(A_1,\ldots,A_r) = \text{false}$, $t(C_1,\ldots,C_r) = \text{true}$.

The following cases are possible:

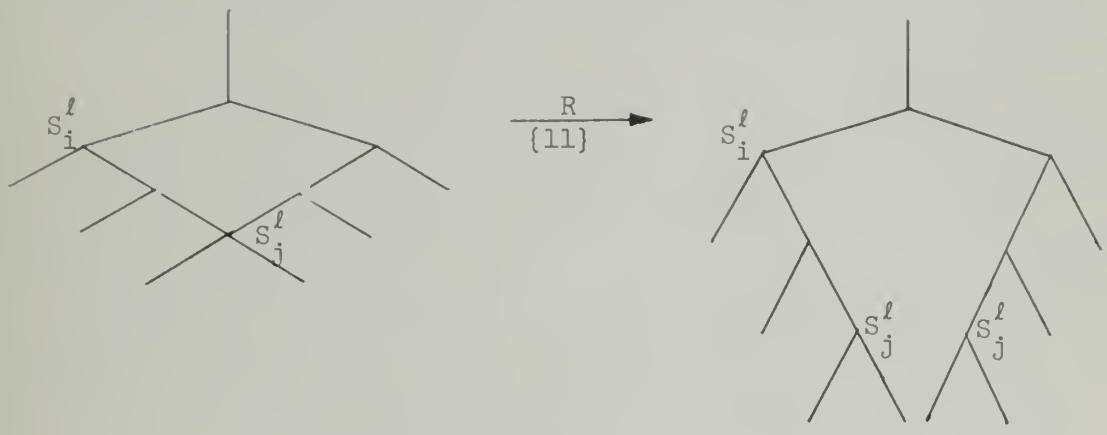


(1) ℓ is the only path that includes S_i^ℓ and S_j^ℓ. In this case conditions (*) and (**) of T6 hold, and T6 can be applied to get rid of the non-executable path.

(2) S_j^ℓ is on both branches of S_i^ℓ, and there is no path that includes S_j^ℓ and not S_i^ℓ. In this case we operate T11 in reverse and then T6 on both branches.



(3) There is at least one path that includes S_j^{ℓ} and not S_i^{ℓ}. In this case we again use T11 in reverse, and then apply T6 on the left subtree.



We may conclude from Theorem 3 that any program that has no loops and contains nonexecutable paths can be transformed to an equivalent program such that each path of its graph is executable. The transformations T6 and T11 are applied as many times as necessary.

Lemma 1

If $\Pi \underset{3}{\Longrightarrow} \Pi'$ then $\Pi \overset{*}{\underset{\{1,2\}}{\Longrightarrow}} \Pi'$.

PROOF:

Assume $S_i = A \leftarrow \Phi B_1 \ldots B_r$. Then $\Pi \underset{3}{\Longrightarrow} \Pi'$ means that in Π' S_i is replaced by $S_i' = C \leftarrow \Phi B_1 \ldots B_r$ and all references to A in the scopes of S_i in all the paths that include S_i are replaced by references to C.

Therefore $\Pi \overset{R}{\underset{2}{\Longrightarrow}} \Pi_1 \underset{1}{\Longrightarrow} \Pi'$

where in Π_1 S_i is replaced by S, S_i'

$S = D \leftarrow \Phi B_1 \ldots B_r$

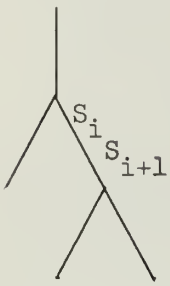
$S_i' = C \leftarrow \Phi B_1 \ldots B_r$

and all references to A in the scopes of S_i in all the paths that include S_i are replaced by references to C. $\Pi_1 \underset{2}{\Longrightarrow} \Pi$ therefore $\Pi \overset{R}{\underset{2}{\Longrightarrow}} \Pi_1$. S is useless in all the paths that include S, therefore $\Pi_1 \underset{1}{\Longrightarrow} \Pi'$.

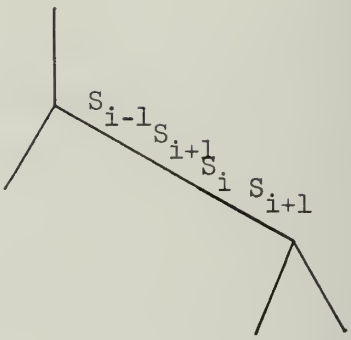
Lemma 2

If $\Pi \underset{4}{\Longrightarrow} \Pi'$ then $\Pi \overset{*}{\underset{\{1,2\}}{\Longrightarrow}} \Pi'$

PROOF:



$\Pi \overset{R}{\underset{1}{\Longrightarrow}} \Pi_1 \underset{2}{\Longrightarrow} \Pi'$ where in Π_1 we insert S_{i+1} between S_{i-1} and S_i,

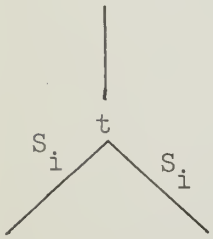


The two statements S_{i+1} are redundant, therefore we can apply T2 and get Π'.

Lemma 3

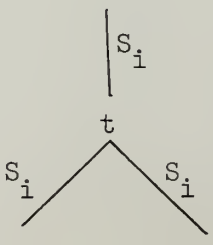
If $\Pi \underset{5}{\Longrightarrow} \Pi'$, then $\Pi \overset{*}{\underset{\{1,2\}}{\Longrightarrow}} \Pi'$.

PROOF:

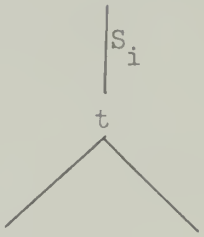


$\Pi \overset{R}{\underset{1}{\Longrightarrow}} \Pi_1 \underset{2}{\Longrightarrow} \Pi'$

We insert S_i before t, which is useless in all the paths that contain it.



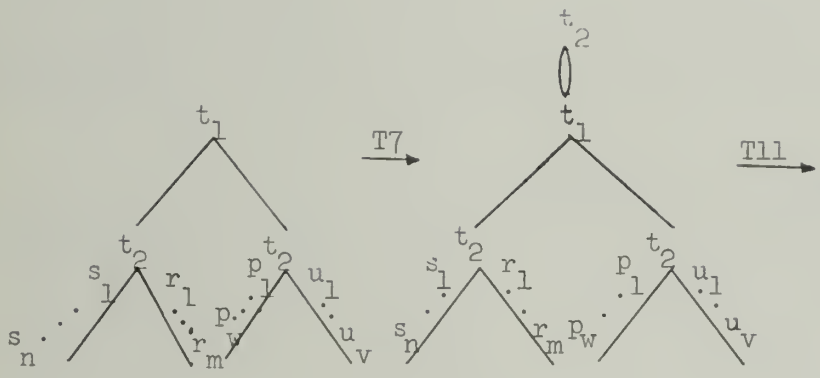
Then we apply T2 twice and we get Π'.

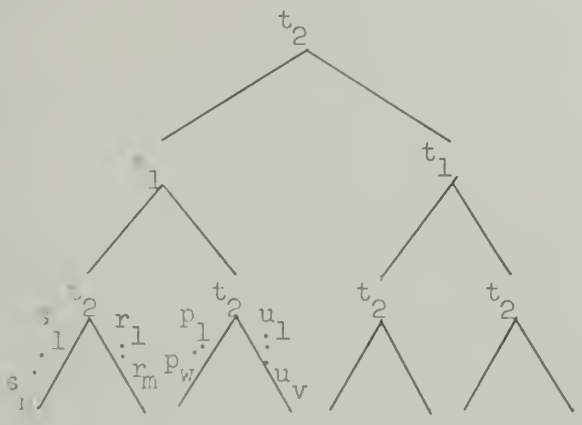


Lemma 4

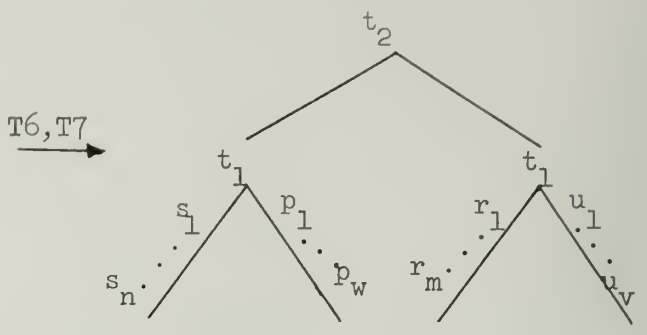
If $\Pi \xrightarrow[8]{} \Pi'$ then $\Pi \xrightarrow[(6,7,11)]{*} \Pi'$.

PROOF:





We operate T6 together with T7 four times:



We will define an enumeration of the paths of a graph corresponding to a program schema. Nonexecutable paths will not be enumerated. The definition of the enumeration will be stated recursively, as follows.

Definition of enumeration

1) look at the root v_i, if it is a leaf - add the path terminated by v_i to the enumeration.

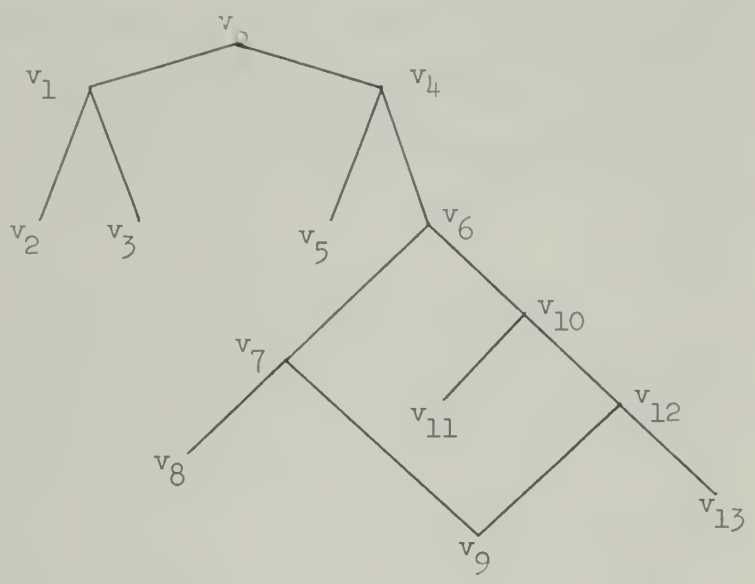
2) otherwise

enumerate the left subgraph

enumerate the right subgraph.

nonexecutable paths are not enumerated.

EXAMPLE 12



The enumeration of the paths (assuming all are executable) will be as follows:

(v_0, v_1, v_2)

(v_0, v_1, v_3)

(v_0, v_4, v_5)

$(v_0, v_4, v_6, v_7, v_8)$

$(v_0, v_4, v_6, v_7, v_9)$

$(v_0, v_4, v_6, v_{10}, v_{11})$

$(v_0, v_4, v_6, v_{10}, v_{12}, v_9)$

$(v_0, v_4, v_6, v_{10}, v_{12}, v_{13})$

Let Π and Π' be two programs in which each path is executable, with the enumerations $\ell_1, \ldots, \ell_n$ and $k_1, \ldots, k_n$ respectively. The enumeration induces a 1-1 correspondence between the paths of Π and Π':

$\ell_1 \rightarrow k_1, \ldots, \ell_n \rightarrow k_n$.

Lemma 5

If Π and Π' are two programs in which corresponding paths have the same tests appearing in the same order, then corresponding paths are consistent.

PROOF:

We will first show that the trees T and T' of Π and Π' respectively are similar, i.e. they have the same structure. By Knuth (6) similarity can be proved by showing that there is a one-to-one correspondence between the nodes of the two trees which preserves the structure, so that if nodes u_1 and u_2 in T correspond respectively to nodes u_1' and u_2' in T', then u_1 is in the left subtree of u_2 iff u_1' is in the left subtree of u_2', and the same holds for right subtrees.

Let $u_1,\ldots,u_n$ and $v_1,\ldots,v_{n'}$ be the enumerations of the nodes of T and T' respectively (with repetitions) induced by the enumerations of the paths. Since corresponding paths have the same tests n=n'. We will show by induction that u_{i+1} is the left descendant of u_i iff v_{i+1} is the left descendant of v_i and u_{i+1} is the right descendant of u_i iff v_{i+1} is the right descendant of v_i.

The case i=1 is trivial.

Assume we proved for 1,2,...,i. Then the subtrees which include the nodes $u_1,\ldots,u_i$ and $v_1,\ldots,v_i$ are similar.

Let u_{i+1} be the first node for which the theorem has not been proven. There are two possible cases:

1) u_{i+1} is the left descendant of u_i. Since all corresponding paths have the same length, v_{i+1} has to be on the same path as v_i, and since the left descendant is the next node to be enumerated, v_{i+1} must be the left descendant of v_i.

2) u_{i+1} is the right descendant of u_i. Since corresponding paths have the same length v_i and v_{i+1} are on the same path in T', thus v_{i+1} is a descendant of v_i. It can not be the left descendant because we assumed that the subtrees which include $u_1,\ldots,u_i$ and $v_1,\ldots,v_i$ are similar. Therefore v_{i+1} is the right descendant of v_i.

Since the trees have the same structure and corresponding paths have the same tests appearing in the same order, it is clear that corresponding paths are consistent.

THEOREM 4

Let Π and Π' be two programs in which each path is executable with the enumerations $k_1,\ldots,k_n$ and $k'_1,\ldots,k'_m$, respectively. Then using the transformations T1,T5,T7,T8 and T11, Π and Π' can be transformed to equivalent programs Ψ and Ψ' respectively with the enumerations $\ell_1,\ldots,\ell_q$ and $\ell'_1,\ldots,\ell'_q$ in which each path is executable, such that corresponding paths in Ψ and Ψ' are consistent and have the same tests.

PROOF:

We will assume that the graphs of Π and Π' are trees. This assumption does not cause any difficulty because T11 may be used in reverse as many times as necessary to transform the graphs to trees.

We will construct two sequences of programs $\Psi_0,\Psi_1,\ldots,\Psi_q$ and $\Psi'_0,\Psi'_1,\ldots,\Psi'_q$ such that $\Psi_0 = \Pi$, $\Psi'_0 = \Pi'$, $\Psi_i \overset{*}{\underset{\{1,5,7,8\}}{\longrightarrow}} \Psi_{i+1}$ $\Psi'_i \overset{*}{\underset{\{1,5,7,8\}}{\longrightarrow}} \Psi'_{i+1}$ and for all i, $1 \leq i \leq q$, the first corresponding i paths of Ψ_i and Ψ'_i are consistent. Then Ψ_q and Ψ'_q will be the programs Ψ and Ψ' of the theorem.

Throughout this proof identical tests checking different values are considered as different tests.

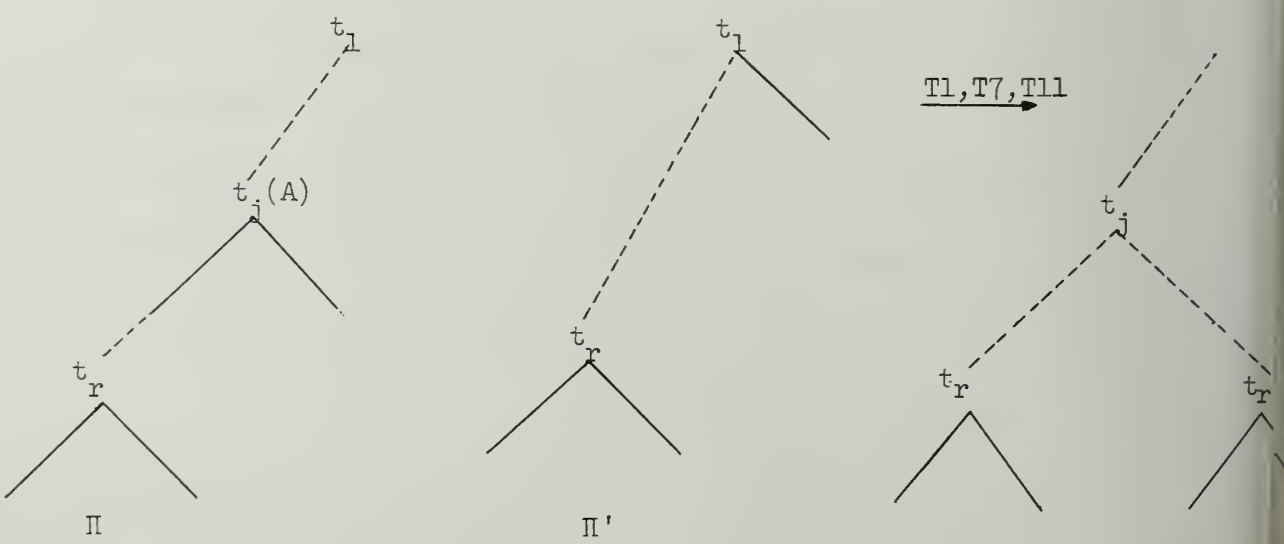
By Lemma 5, if corresponding paths have the same tests appearing in the same order, then corresponding paths are consistent. So in each stage of the process we apply transformations T1,T5,T7,T8 so that corresponding paths will have the same tests appearing in the same order. Useless unconditional tests will be eliminated by T7 before applying the above transformations.

Ψ_1 and Ψ_1' will be constructed as follows:

First we eliminate all useless unconditional tests by applying T7. Then

1) If the tests in k_1 and k_1' are $t_1,\ldots,t_r$ and they appear in the same order in both paths, then $\Psi_1 = \Pi$ and $\Psi_1' = \Pi'$.

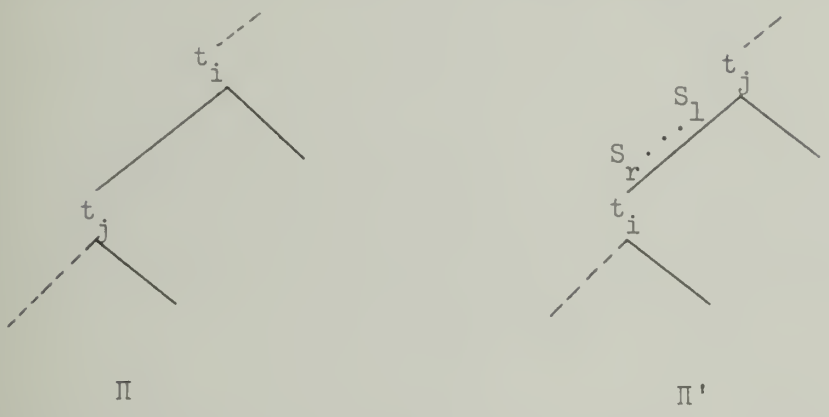
2) If the tests in both k_1 and k_1' appear in the same order, but there is a test t_j in k_1 that does not appear in k_1', we use T1 reversed as many times as necessary to insert statements such that $v^{k_1}(A) = v^{k_1'}(A)$, and then we use the reverse of T7, T11 to insert $t_j(A)$ in k_1'. We get an equivalent program.



We operate the reverse of T1,T7 and T11 as many times as necessary on both Π and Π' so that both paths will have the same tests.

$$\Pi \xrightarrow[\{1,7,11\}]{*} \Psi \qquad \Pi' \xrightarrow[\{1,7,11\}]{*} \Psi'.$$

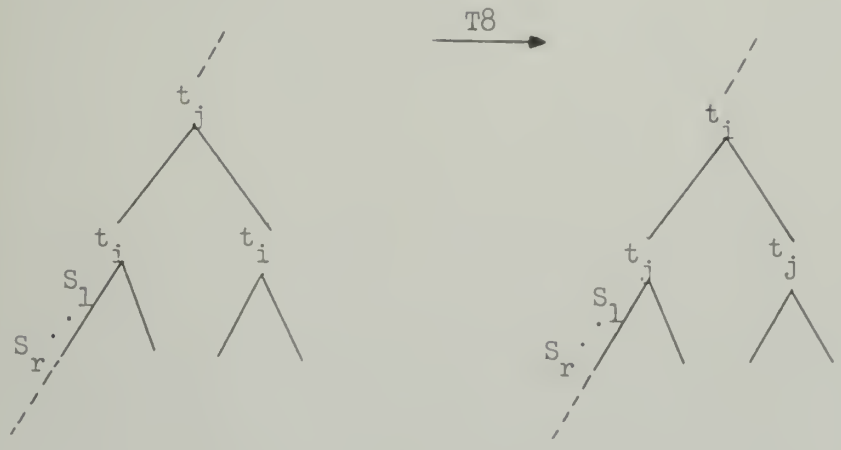
3) Assume the tests appear in a different order and let t_i and t_j be two adjacent tests appearing in a different order in Π and Π'.



Π $\qquad\qquad$ Π'

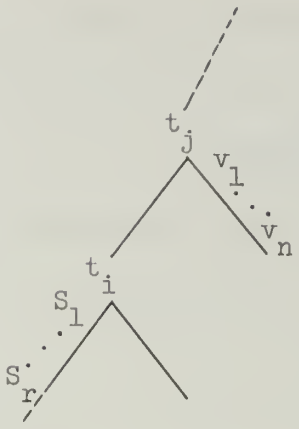
We will use the reverse of T5 as many times as necessary to move the statements $S_1,\ldots,S_r$ after t_i so that T8 can be operated.

(a) if t_i appears on the right branch of t_j in Π' right after t_j we will operate T8 on Π'.



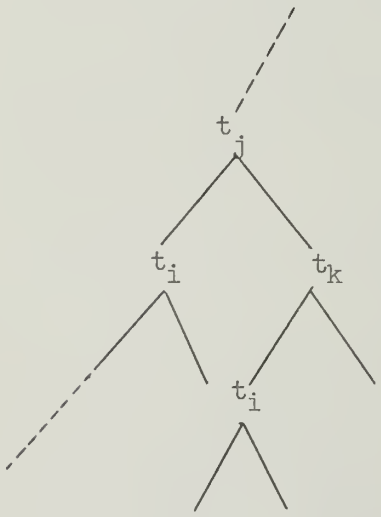
The obtained program will have t_i and t_j in the same order as in Π.

(b) Otherwise



we will operate the reverse of T7 to insert t_i on the right branch of t_j (if v_n is a STOP statement, we will insert t_i before v_n), and then use the reverse of T5 as many times as necessary to move all the assignment statement after t_i. Now T8 can be applied.

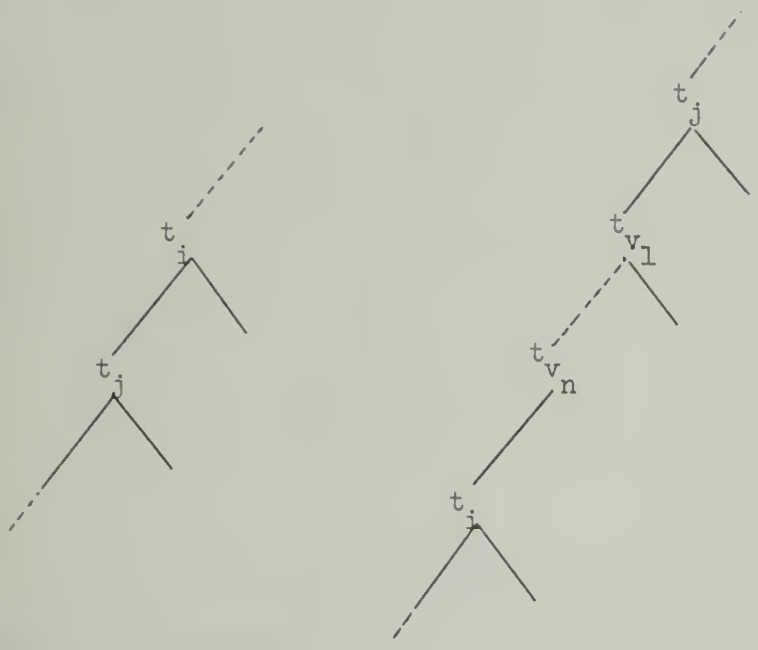
(c) If t_i appears on the right branch of t_j but there are test statements between t_i and t_j, we will use T7 to insert t_i on the right branch of t_k and then use T8. Now we apply the same procedure as in case (a).



The above process will be applied to each pair of t_i and t_j that appear in a different order in the two paths.

$$\Pi \xrightarrow[\{1,5,7,8\}]{*} \Psi_1 \qquad\qquad \Pi' \xrightarrow[\{1,5,7,8\}]{*} \Psi'_1$$

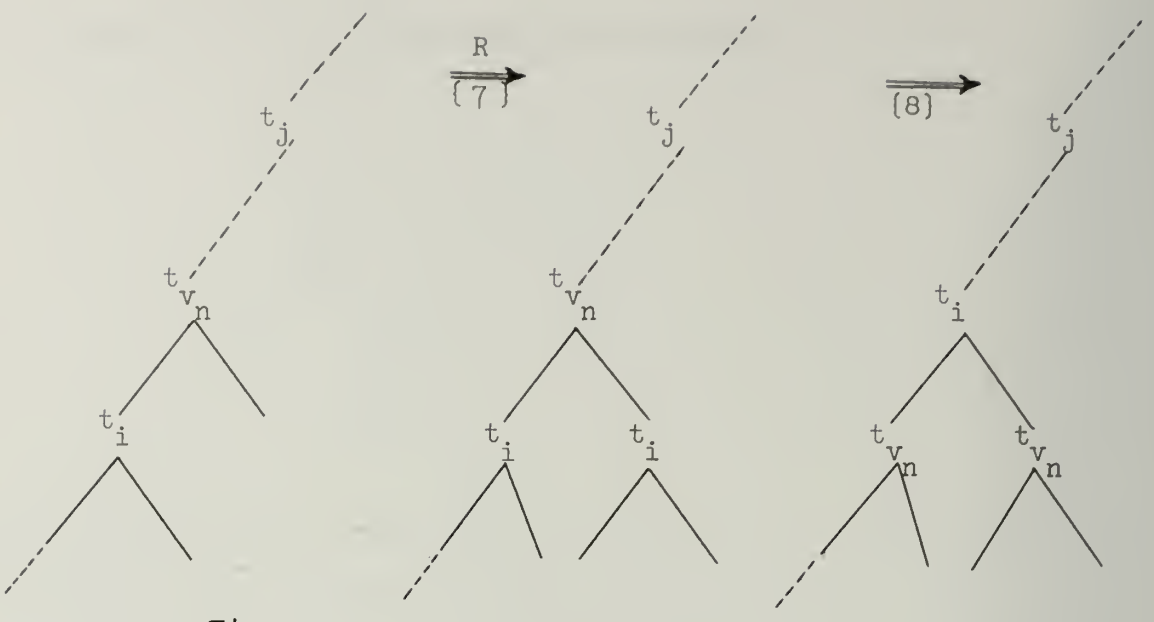
4) The same as in 3) but t_i and t_j are not adjacent in Π'.



Π Π'

In this case we will use T7 to insert t_i on the right branch of t_{v_n} and then apply T8. We might have to use T5 to move assignment statements on both branches of t_{v_n} so that T8 can be applied. We will use the same procedure as in case 3).

We will repeat this process for all $t_{v_n},\ldots,t_{v_1}$.



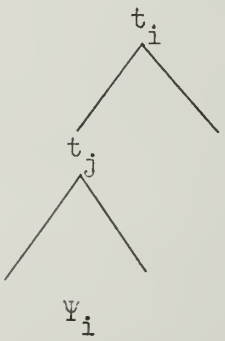
Π'

$\Pi' \xrightarrow[\{5,7,8\}]{*} \Psi_1'$.

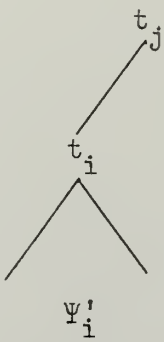
We conclude from 1)-4) that $\Pi \xrightarrow[\{1,5,7,8\}]{*} \Psi_1$ $\Pi' \xrightarrow[\{1,5,7,8\}]{*} \Psi_1'$.

Assume that we have constructed $\Psi_0,\ldots,\Psi_i$ and $\Psi_0',\ldots,\Psi_i'$ such that for the first i paths of Ψ_i and Ψ_i' - corresponding paths are consistent. We would like to construct Ψ_{i+1} and Ψ_{i+1}'.

Let the tests of ℓ_{i+1} and ℓ'_{i+1} be $t_1,\ldots,t_r$ and $t_1',\ldots,t_q'$ respectively. The cases 1),2) are as before. For 3),4) assume first that t_j is on the left branch of t_i in Ψ_i,



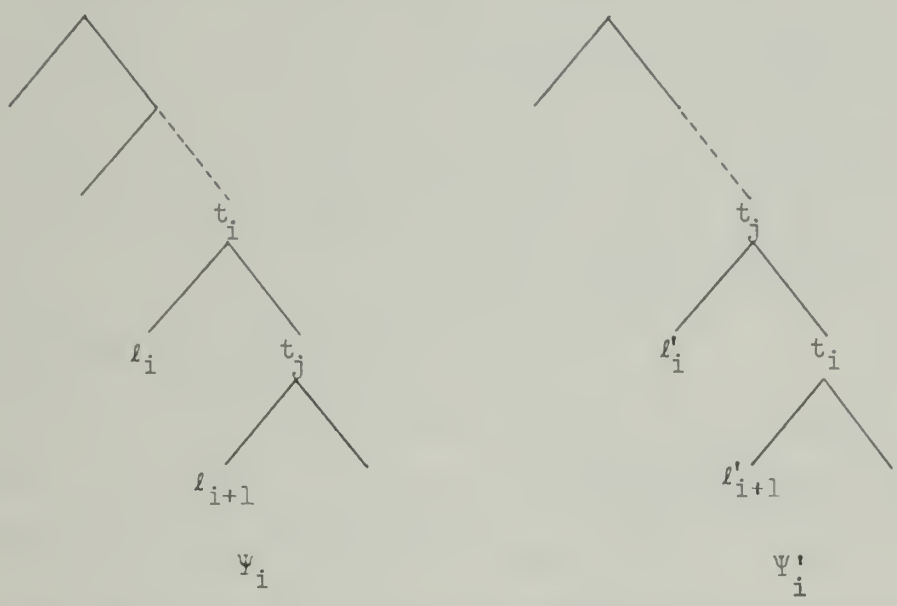
Ψ_i



Ψ_i'

then proof is as in 3) above.

If t_j is on the right branch of t_i



we observe that this case is impossible because ℓ_i and ℓ_i' include the same tests appearing in the same order.

4) is proved in the same way.

Remark: We observe that in Ψ and Ψ' no path has useless unconditional tests.

THEOREM 5

Let Π and Π' be two programs with the enumerations $\ell_1,\ldots,\ell_n$ and $k_1,\ldots,k_n$ respectively, in which

i) each path is executable.

ii) corresponding paths are consistent and have the same tests.

iii) all the statements of the programs appear on the graph (i.e. there are no unreferenced blocks).

iv) no path has useless unconditional tests.

Then

$$\forall\, i,\quad 1 \leq i \leq n,\quad D_i(\Pi) = D_i(\Pi') \quad \text{iff} \quad \Pi \xrightarrow[\{3,4,5,10,11\}]{*} \Pi' .$$

$D_i(\Pi)$ denotes the dag $D_{\ell_i}(\Pi)$ $1 \leq i \leq n$, and $D_i(\Pi')$ denotes the dag $D_{k_i}(\Pi')$, $1 \leq i \leq n$.

PROOF:

1) The "if" part is simple and was discussed when we listed the transformations.

2) We will show that if $\ell_1,\ldots,\ell_n$ are the paths of Π, $k_1,\ldots,k_n$ are the paths of Π', and $\forall\, 1 \leq i \leq n$ $D_i(\Pi) = D_i(\Pi')$ then $\Pi \xrightarrow[\{3,4,5,10,11\}]{*} \Pi'$.

If the dags are identical then the input sets of Π and Π' are identical because the inputs are the leaves of the dags. Let $S_1^{\ell_i},\ldots,S_{n_i}^{\ell_i}$ be the sequence of assignment statements of path ℓ_i in program Π. $T_1^{k_i},\ldots,T_{n_i}^{k_i}$ is the sequence of assignment statements of path k_i in program Π'. The two sequences have the same length because each statement corresponds to a node of D_i, and we assumed that the dags are identical.

Renumber the assignment statements $S_1,\ldots,S_m$ of Π by eliminating repetitions from the sequence $S_1^{\ell_1},\ldots,S_{r_1}^{\ell_1},..,S_1^{\ell_n},\ldots,S_{r_n}^{\ell_n}$. We shall define inductively a sequence of programs Π_0, $\Pi_1,\ldots,\Pi_m$ such that $\Pi_0 = \Pi'$, $\Pi_m = \Pi$ and $\Pi_j \xrightarrow[\{3,4,5,10,11\}]{*} \Pi_{j+1}$. Assume inductively that 1) $\Pi' \xrightarrow[\{3,4,5,10,11\}]{*} \Pi_j$, 2) corresponding paths in Π' and Π_j are consistent and have the same tests, 3) If U_1^j, $U_2^j,\ldots,U_m^j$ are the assignment statements of Π_j numbered without repetitions using the same method as was used to number the statements $S_1,\ldots,S_m$ of Π, then $U_1^j = S_1$, $U_2^j = S_2,\ldots,U_j^j = S_j$.

Clearly the assumptions 1-3 are valid for $j=0$.

Let Π_j be given and assume S_{j+1} is $A \leftarrow \varphi A_1 \ldots A_k$ and lies on path ℓ_i. Because $D_i(\Pi_j) = D_i(\Pi)$ there is an assignment statement $U_v^j = B \leftarrow \varphi A_1 \ldots A_k$ that has the same node in $D_i(\Pi_j)$ that S_{j+1} has in $D_i(\Pi)$. We see that $v > j$ because U_v^j is not in the sequence $U_1^j, \ldots, U_j^j$. None of the $A_1, \ldots, A_k$ is defined by $U_{j+1}^j, \ldots, U_v^j$ because otherwise $D_i(\Pi)$ would not be equal to $D_i(\Pi_j)$.

Let S_r be the last statement in path ℓ_i for which the corresponding statement in Π_j appears in the same position. Since the enumeration is without repetitions S_r may be different from S_j.

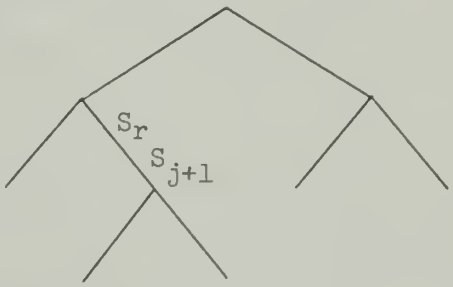
We will distinguish between two cases:

(a) in Π, S_{j+1} is the next statement after S_r.

(b) S_{j+1} is the next assignment statement but there are one or more test statements between S_r and S_{j+1}.

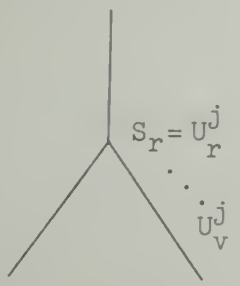
S_r might be null in the initial step of $\Pi_j = \Pi'$.

(a) S_{j+1} follows S_r in Π.



The following cases are possible for U_v^j.

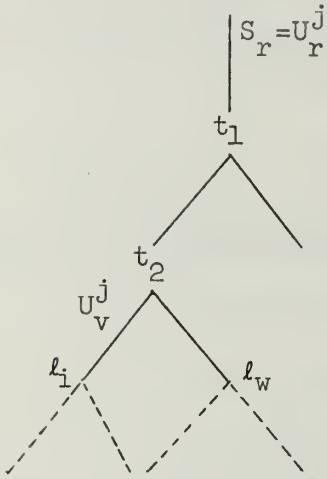
1) There is no test between $U_r^j = S_r$ and U_v^j in Π_j, and there is no path that includes U_v^j but not U_r^j.



We can apply T4 as many times as necessary to move U_v^j right after S_r. T4 can be applied because we proved that none of $A_1, \ldots, A_k$ is defined by $U_{j+1}^j, \ldots, U_v^j$.

By T3 we change B to A. We get a program Π_{j+1}, $\Pi_j \xrightarrow[\{3,4\}]{*} \Pi_{j+1}$ and $U_1^j = S_1, \ldots, U_{j+1}^{j+1} = S_{j+1}$

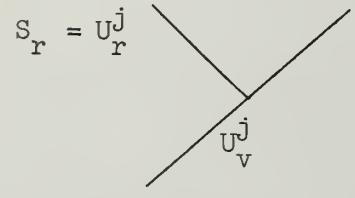
2) In Π_j there is one or more tests between $S_r = U_r^j$ and U_v^j, but there is no path that includes U_v^j but not U_r^j.



(i) If U_v^j is on the left path of the last test that separates U_r^j and $U_v^j(\ell_i)$ then it must be also on the right path of this test (ℓ_w) because the corresponding path of ℓ_w in Π contains S_{j+1} and the dags of Π and Π_j are equal. By applications of T4 and T5 several times we can move U_v^j to be right after U_r^j, using the argument of 1). By T3 we change B to A.

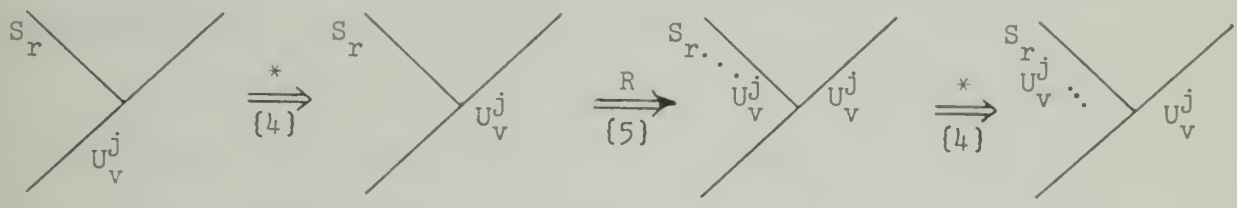
(ii) If U_v^j is on the right path of the last test that separates U_r^j and U_v^j, then by the above argument there must be a statement $U_w^j = U_v^j$ on the left path of that test. But because the left path precedes the right path in the enumeration and $U_1^j = S_1, \ldots, U_r^j = S_r$, this case is impossible.

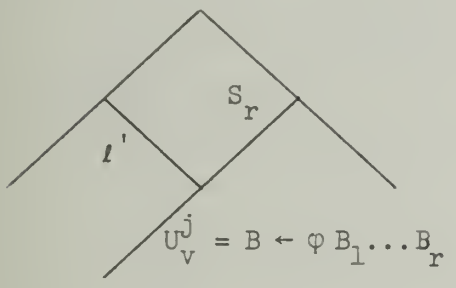
3) There is a path that contains U_v^j and does not contain S_r, and there is no test between S_r and U_v^j.



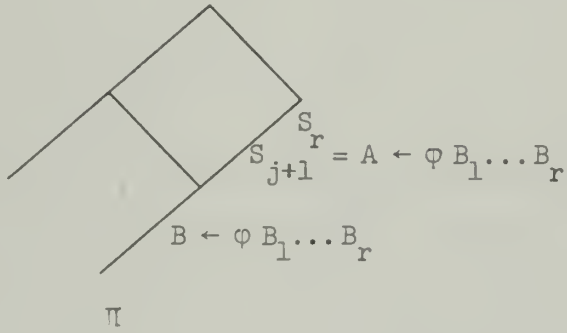
(i) If S_r is on the path left of the merging point, we use T4 to move U_v^j to the merging point and then use the reverse of T5, and again T4. Increasing the number

of statements by using the reverse of T5 does not cause any difficulty, because the number of statements along all the paths is equal in Π and Π'.



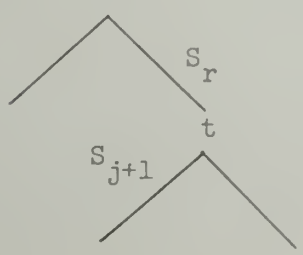


(ii) The case in which S_r is on the path right of the merging point is impossible because we proved already for the left path ℓ', therefore Π has the form



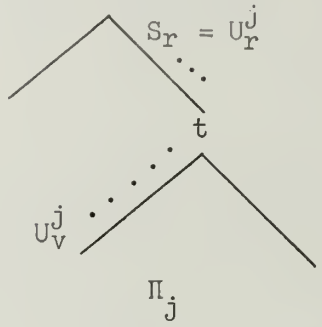
and there must be another statement U_v^j in Π_j that gives rise to a node labeled by φ with descendants $B_1, \ldots, B_r$. This statement must be before the merging point, otherwise we apply the same argument. We use the same procedure as in 1).

(b) In Π there are one or more test statements between S_r and S_{j+1}.



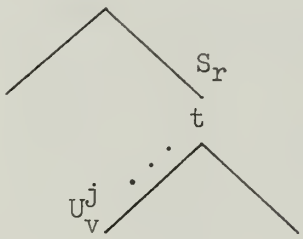
Assume there is one test t between S_r and S_{j+1}. Because we assumed that corresponding paths in Π' and Π_j are consistent and have the same tests,

and also corresponding paths in Π' and Π are consistent and have the same tests, also corresponding paths in Π and Π_j are consistent and have the same tests. Therefore in Π_j there must be a test statement t checking the same values and there is no test statement between $U_r^j = S_r$ and t.



We use T5 reverse as many times as necessary, so in Π_{j+1} t will appear right after S_r.

$$\xrightarrow[\{5\}]{R}$$



Then we proceed with similar arguments to those in case (a). We use similar arguments when there is more than one test between S_r and S_{j+1}.

We might conclude from (a) and (b) that the desired program Π_{j+1} is obtained. In each stage we might have to flip blocks by T10 so that the blocks will appear in the same order in Π and Π_{j+1}.

$$\Pi_j \xrightarrow[\{3,4,5,10,11\}]{*} \Pi_{j+1} \quad \text{thus} \quad \Pi' \xrightarrow[\{3,4,5,10,11\}]{*} \Pi_{j+1}$$

and because none of these transformations operate on tests, corresponding paths in Π' and Π_{j+1} are consistent and have the same tests.

DEFINITION

A program is reduced if no executable path of the program has useless or redundant statements and the program has no unreferenced blocks.

The dags of a reduced program have the following properties:

1) No dag in D(Π) has roots which are not distinguished.

2) No two nodes in a dag have identical direct descendants.

THEOREM 6

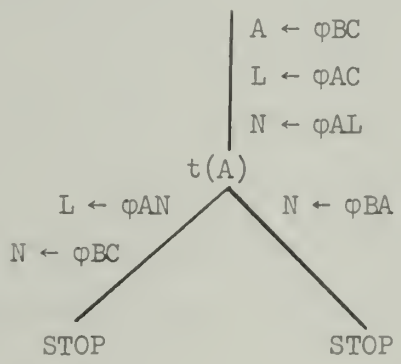
Any program Π can be transformed to an equivalent reduced program Π' using no larger set of transformations than {1,2,4,5,9,11}.

The following example shows how the transformations are used to reduce a given program.

EXAMPLE 13

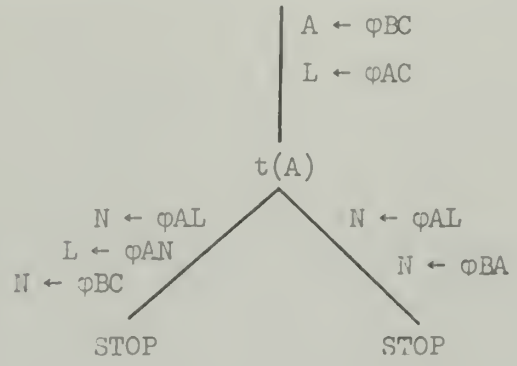
Let Π be $\Pi = (G_P, \{B,C\}, \{L,N\})$

G_P:

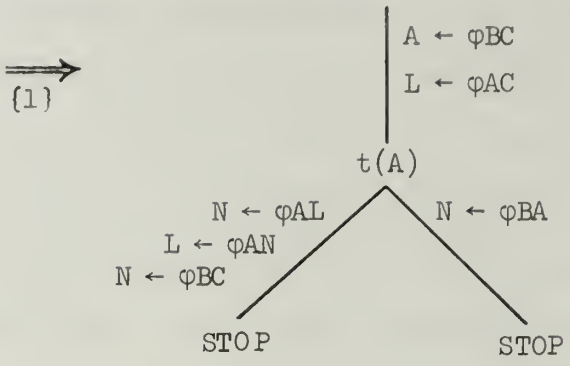


The statement N ← φAL is useless in the right path. To remove the statement we first use the reverse of T5

$\Pi \xrightarrow[\{5\}]{R}$



now T1 might be used to remove the useless statement:



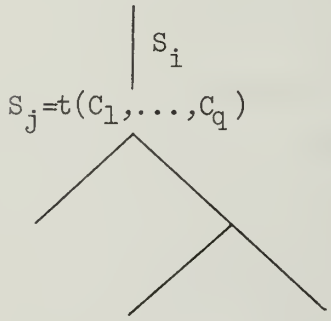
The obtained program is reduced.

PROOF:

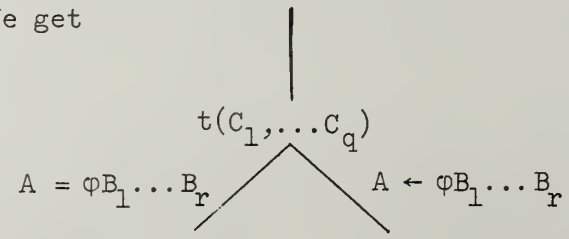
Assume $S_i^k : A \leftarrow \varphi B_1 \ldots B_r$ is a useless statement in an executable path k of the program Π.

(a) If S_i is useless in all paths containing it, then we operate T1 to remove S_i.

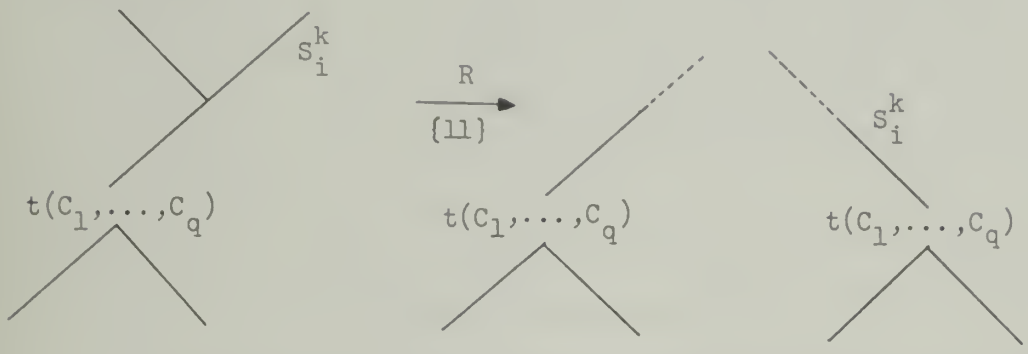
(b) If there exists at least one path ℓ such that S_i is not useless in ℓ, let $S_j^k = t(C_1,\ldots,C_q)$ be the first test statement after S_i^k.



i) Assume there is no path that includes $t(C_1,\ldots,C_q)$ and does not include $S_i^k \cdot S_i^k$ is useless in path k, so A is not referenced by any statement $S_{i+1}^k,\ldots,S_j^k$. Thus we might use T4 as many times as necessary so that S_i^k appears before the test statement. Now we use the reverse of T5. We get

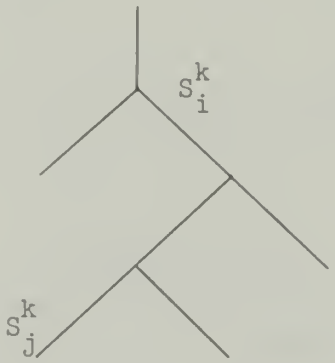


ii) Assume there is a path that includes $t(C_1,\ldots,C_q)$ and does not include S_i^k.

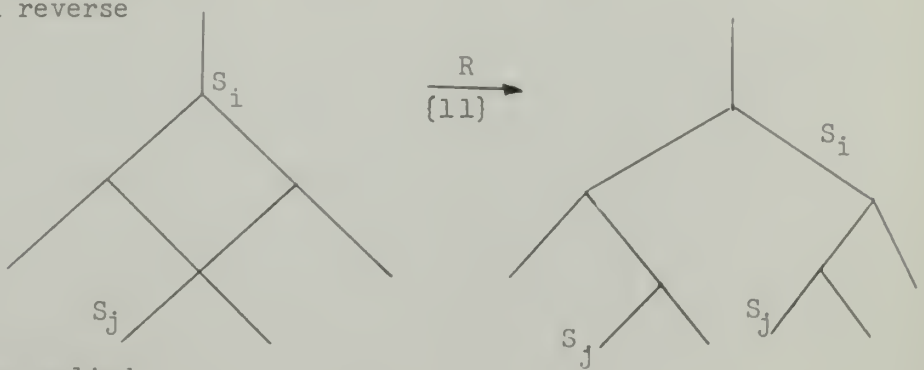


We operate T11 in reverse and proceed as in case i). We repeat the process of i) and ii) above as many times as necessary till S_i^k will be useless in all paths that include it and T1 can be applied.

Assume $S_i^k : A \leftarrow \varphi D_1 \ldots D_r$ and $S_j^k : B \leftarrow \varphi D_1 \ldots D_r$ are redundant in path k. If there is no path that includes S_j^k and does not include S_i^k and also there is no path ℓ that includes S_i^ℓ and S_j^ℓ but they are not redundant in that path, we can operate T2 and remove S_j^k.

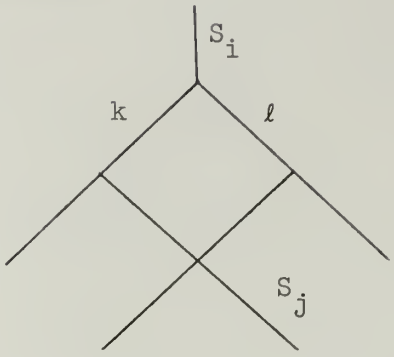


If there is a path that includes S_j^k and does not include S_i^k we operate T11 in reverse

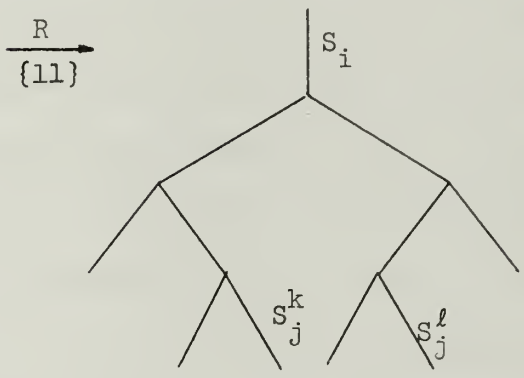


and now T2 can be applied.

Assume there is a path ℓ that includes S_i^{ℓ} and S_j^{ℓ} and they are not redundant in that path



then we again operate T11 in reverse



and now T2 can be applied to remove S_j^k from path k.

Unreferenced blocks can be removed by applying T9. So

$$\Pi \xrightarrow[\{1,2,4,5,9,11\}]{*} \Pi'.$$

Lemma 6

Let Π be a program in which each path is executable and E is a well-formed expression over Θ and Σ such that E is in $v^{\ell}(\Pi)$ for some path ℓ or $E = v^{\ell}(C_i)$ where $t^{\ell}(C_1,\ldots,C_n)$ is a test statement in path ℓ. $E \notin \Sigma$. Let E_1 be a well-formed expression, $E_1 \notin \Sigma$, E_1 is a subexpression of E, then there is a statement $A \leftarrow \varphi A_1 \ldots A_k$ in path ℓ such that the value of A computed at that time is E_1.

PROOF:

Because of the properties of well-formed expressions, there is a unique way to write an expression E as $\varphi E_1 \ldots E_k$ where $E_1, \ldots, E_k$ are subexpressions. The proof follows from the definition of the value of a variable.

Lemma 7

Let Π and Π' be two equivalent, reduced programs in which all paths are executable and corresponding paths are consistent and have the same tests, and let ℓ and k be two corresponding paths of Π and Π' respectively.

Then the assignment statements of ℓ and k are in one-to-one correspondence such that if $S_j^{\ell} : A \leftarrow \varphi A_1 \ldots A_r$ and $U_q^k : B \leftarrow \Psi B_1 \ldots B_s$ correspond, then $v_j^{\ell}(A) = v_q^k(B)$.

PROOF:

Let $S_j^{\ell} : A \leftarrow \varphi A_1 \ldots A_r$ be an assignment statement of ℓ such that $v_j^{\ell}(A) = E$. Π is reduced, therefore one of the following two conditions must hold.

1) E is a subexpression of an expression in $v^{\ell}(\Pi)$. In this case since ℓ and k are consistent and $\Pi = \Pi'$, $v^{\ell}(\Pi) = v^k(\Pi')$. Therefore E is also a subexpression of an expression in $v^k(\Pi')$.

2) E is a subexpression of an expression $v^{\ell}(C_i)$ such that $t^{\ell}(C_1, \ldots, C_n)$ is a test statement in path ℓ. In this case since ℓ and k are consistent, corresponding, and have the same tests, there exists a test statement $t^k(D_1, \ldots, D_n)$ such that $v^k(D_i) = v^{\ell}(C_i)$. Thus E is also a subexpression of $v^k(D_i)$.

By Lemma 6 there is at least one q such that $v^k_q(B) = E$. We will prove that q is unique. Suppose there are q_1 and q_2 for some j. (If there is more than one such j select the first one and if for that j there are more than two q's, select the first two.) Let

$$U^k_{q_1} : B \leftarrow \Psi_1 B_1 \ldots B_r$$

$$U^k_{q_2} : C \leftarrow \Psi_2 C_1 \ldots C_r$$

$v^k_{q_1}(B) = v^k_{q_2}(C)$ thus $\Psi_1 = \Psi_2$ and $v^k_{q_1}(B_i) = v^k_{q_2}(C_i)$. $1 \leq i \leq r$.

Because of the selection of j, q_1 and q_2 $B_i = C_i$ $1 \leq i \leq r$. Thus we can apply T2 to Π', and this contradicts the assumption that Π' is reduced.

THEOREM 7

Let Π and Π' be two reduced programs in which

1) each path is executable

2) corresponding paths are consistent and have the same tests.

Then

$$\Pi \equiv \Pi' \quad \text{iff} \quad D_i(\Pi) = D_i(\Pi') \quad \forall i, 1 \leq i \leq n$$

$\ell_1, \ldots, \ell_n$ and $k_1, \ldots, k_n$ are the enumerations of Π and Π' respectively.

PROOF:

1) If for all i, $1 \leq i \leq n$ $D_i(\Pi) = D_i(\Pi')$, by theorem 5 $\Pi \xrightarrow[\{3,4,5,10,11\}]{*} \Pi'$ and because transformations preserve equivalence $\Pi \equiv \Pi'$.

2) If $\Pi \equiv \Pi'$ and there is at least one i for which $D_i(\Pi) \neq D_i(\Pi')$, there is a node N in $D_i(\Pi)$ which is not included in

$D_i(\Pi')$, but all its descendants are present. (simplest case - where the descendants are the input variables which are the same for both programs.) This contradicts Lemma 7 because we showed that the assignment statements are in one-to-one correspondence $A \leftarrow \varphi A_1 \ldots A_r \longleftrightarrow B \leftarrow \psi B_1 \ldots B_r$ such that $v_j^i(A) = v_k^i(B)$. Thus $\varphi = \psi$ and $v_j^i(A_q) = v_j^i(B_q)$ $1 \leq q \leq r$. Since each node of the dag is associated with an assignment statement, the contradiction follows.

THEOREM 8

Let Π and Π' be two loop-free programs. Then

$$\Pi \equiv \Pi' \quad \text{iff} \quad \Pi \xrightarrow[\{1,2,6,7,9,10,11\}]{*} \Pi'.$$

PROOF:

1) "if" is trivial because transformations preserve equivalence.

2) By Theorem 3

$$(*) \quad \Pi \xrightarrow[\{6,11\}]{*} \Pi_1 \qquad \Pi' \xrightarrow[\{6,11\}]{*} \Pi_1'$$

where Π_1, Π_1' are programs in which each path is executable.

$\Pi_1 \equiv \Pi \equiv \Pi' \equiv \Pi_1'$.

By Theorem 4

$$\Pi_1 \xrightarrow[\{1,5,7,8,11\}]{*} \Pi_2 \qquad \Pi_1' \xrightarrow[\{1,5,7,8,11\}]{*} \Pi_2'$$

such that corresponding paths in Π_2 and Π_2' are consistent and have the same tests and each path in Π_2 and Π_2' is executable. By lemmas 3 and 4

$$(**) \quad \Pi_1 \xrightarrow[\{1,2,6,7,11\}]{*} \Pi_2 \qquad \Pi_1' \xrightarrow[\{1,2,6,7,11\}]{*} \Pi_2' .$$

$\Pi_1 \equiv \Pi_1'$ therefore $\Pi_2 \equiv \Pi_2'$.

We transform Π_2 and Π_2' to equivalent reduced programs Π_3 and Π_3', respectively. By Theorem 6

$$\Pi_2 \xrightarrow[\{1,2,4,5,9,11\}]{*} \Pi_3 \qquad \Pi_2' \xrightarrow[\{1,2,4,5,9,11\}]{*} \Pi_3' .$$

By Lemmas 2, 3

$$(***)\quad \Pi_2 \xrightarrow[\{1,2,9,11\}]{*} \Pi_3 \qquad \Pi_2' \xrightarrow[\{1,2,9,11\}]{*} \Pi_3' .$$

$\Pi_2 \equiv \Pi_2'$ thus $\Pi_3 \equiv \Pi_3'$.

By Theorem 7 since Π_3 and Π_3' are reduced programs in which each path is executable and corresponding paths are consistent and have the same tests

$$\forall i,\ 1 \leq i \leq n \quad D_i(\Pi_3) = D_i(\Pi_3')$$

where $\ell_1,\ldots,\ell_n$ and $k_1,\ldots,k_n$ are the enumerations of Π_3 and Π_3' respectively.

By Theorem 5 $\Pi_3 \xrightarrow[\{3,4,5,10,11\}]{*} \Pi_3'$.

By Lemmas 1-3 $\Pi_3 \xrightarrow[\{1,2,10,11\}]{*} \Pi_3'$.

By using (***) $\Pi_2 \xrightarrow[\{1,2,9,10,11\}]{*} \Pi_2'$.

By using (**) $\Pi_1 \xrightarrow[\{1,2,6,7,9,10,11\}]{*} \Pi_1'$

and by using (*) $\Pi \xrightarrow[\{1,2,6,7,9,10,11\}]{*} \Pi'$.

This shows that {T1,T2,T6,T7,T9,T10,T11} form a complete set of transformations.

DEFINITION

If J is a complete set of transformations, but no proper subset of J is complete, then J is called an irredundant set of transformations.

THEOREM 9

J = {T1,T2,T6,T7,T9,T10,T11} form an irredundant set of transformations.

PROOF:

By Theorem 8 the set J is complete. We will prove the theorem by showing that each of the transformations in J has to be in the irredundant set.

Let S be the set $\{T_j\}_{j=1}^{11}$.

a) For showing that T1 has to be in J we will use Aho's and Ullman's example (1).

$\Pi_1 = (P, \{B,C\}, A)$ $\qquad$ $\Pi_1' = (P', \{B,C\}, A)$

P: $A \leftarrow \Psi BC$ $\qquad$ P': $A \leftarrow \varphi BC$

$A \leftarrow \varphi BC$ $\qquad$ STOP

STOP

$\Pi_1 \underset{1}{\Longrightarrow} \Pi_1'$, but there is no sequence of transformations from S - {T1} that can get rid of the first statement. The only transformation that can remove an assignment statement from an executable path is T2, and T2 eliminates redundant statement. Here the expressions ΨBC and φBC are different and therefore are irredundant. Adding new paths to the program by T7 and T11 will not help because there will still be at least one path that contains the useless statement.

b) We will show that T2 is in J by using again an example of Ullman and Aho (1).

Π_2 = (P, {C,D}, {A,B})

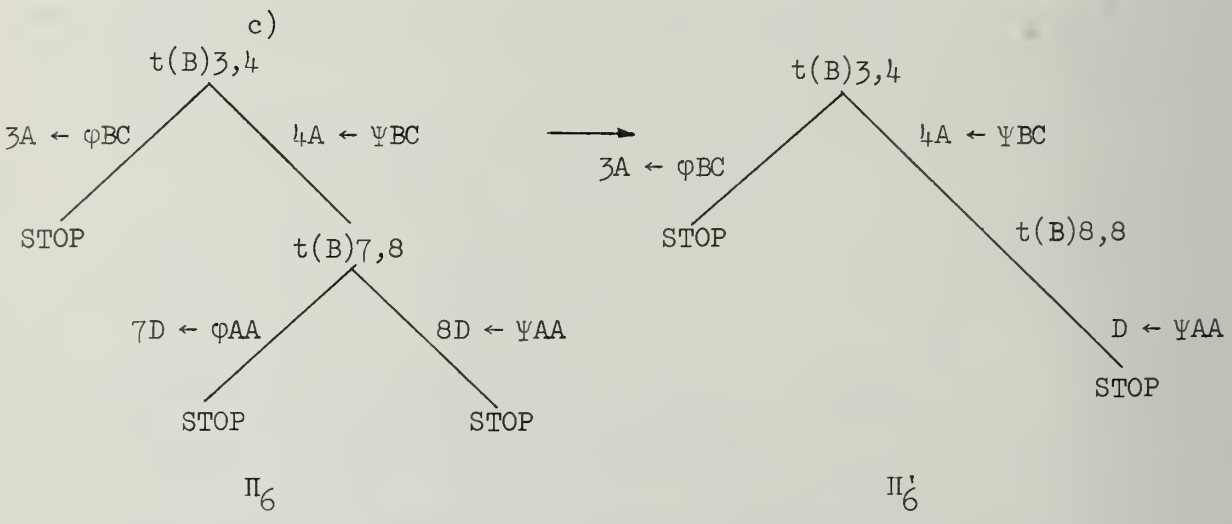
P:
A ← φCD
B ← ΨAA
A ← φCD
STOP

Π_2' = (P', {C,D}, {A,B})

P':
A ← φCD
B ← ΨAA
STOP

$\Pi_2 \overset{2}{\Longrightarrow} \Pi_2'$, but no combination of transformations from S - {T2} can get rid of the third statement. That is because none of the statements is useless, and therefore T1 can not be applied. Also, if we introduce new paths to the program by T7 and T11 there will still be at least one path that includes the redundant statements. Thus $\Pi_2 \overset{S - \{T2\}}{\not\Longrightarrow} \Pi_2'$.

c)

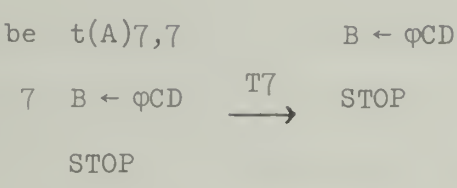


Π_6 Π_6'

No combination of transformations of S - {T6} can change the conditional test t(B)7,8 in Π_6 to an unconditional test and thus eliminate the nonexecutable path. That is because none of the transformations operate on transfer addresses of tests and decide if a conditional

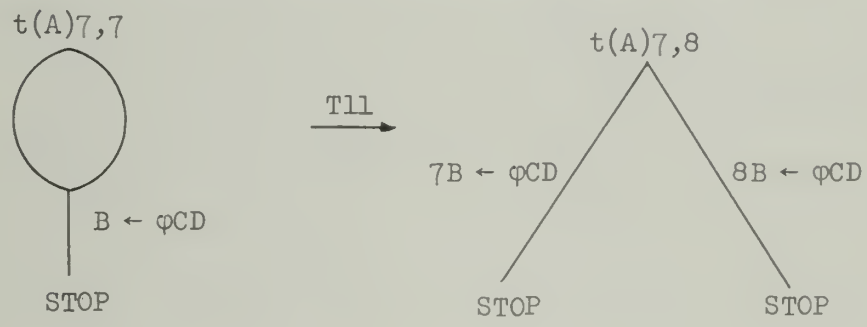
test can be made unconditional. Introducing new paths to the program by T7 and T11 will not help, since the above path will remain nonexecutable.

d) Let Π_7 be

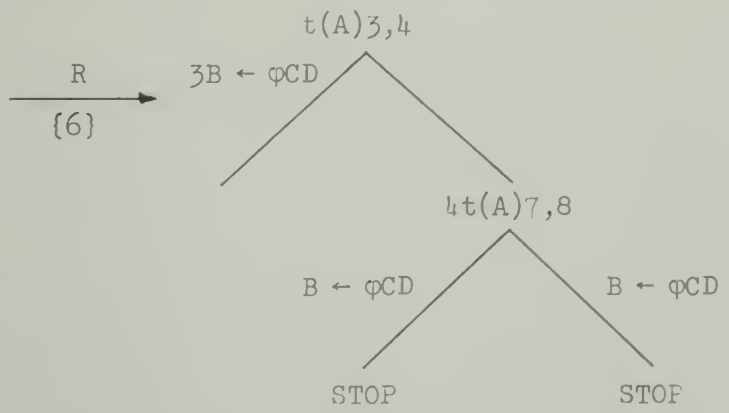


We will show that no combination of transformations from S - {T7} can eliminate the useless test.

If we look at the transformations that operate on tests- T6, T8, T11 then if we use T11 reverse we get



and t(A) can not be eliminated from the obtained program, only by using T11 again and T7. Introducing other new paths can only be done by using T7 and T11, or by introducing nonexecutable paths by T6 reverse. If we introduce a test by T6 reverse



We can eliminate the original t(A) by using again T6 but we have now a new test that can not be eliminated. T8 operates on tests but can not eliminate useless tests.

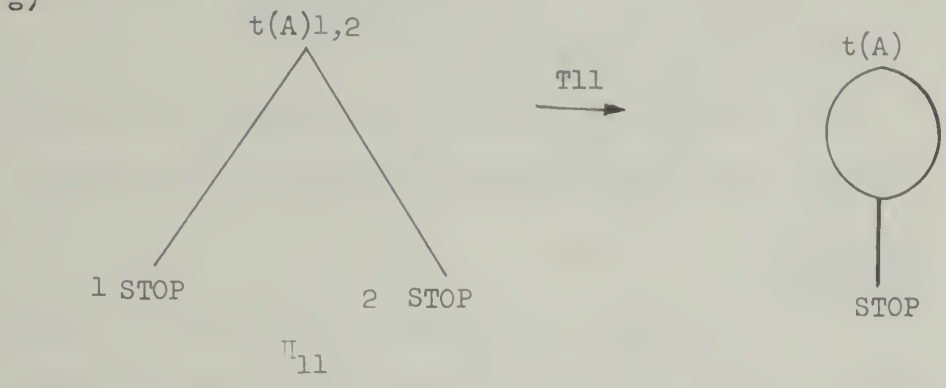
e) For the case of T9, removing of unreferenced blocks, we observe that T1-T5 manipulate assignment statements and do not operate on the whole block, T6-T8 operate on tests, T10 flip blocks and T11 merge identical subgraphs, but cannot remove unreferenced blocks because they do not appear on the graph. Therefore no sequence of transformations from S - {T9} can get rid of unreferenced blocks.

f)

	t(A) 1,2			t(A) 1,2
1	A ← φBC		2	A ← ΨBC .
	STOP			STOP
2	A ← ΨBC	T10 →	1	A ← φBC
	STOP			STOP
	Π_{10}			Π'_{10}

No sequence of transformations from S - { T10} can transform Π_{10} to Π'_{10}. That is because no sequence of transformations can change the relative positions of different blocks in the program. T4 change the relative order of assignment statements but not of blocks. Also T11 cannot merge these blocks because they are not equal. Introducing new tests will not help since the transfer addresses of different tests might be changed but the blocks themselves will not be removed.

g)



No sequence of transformations from S - {T11} can merge these subgraphs. That is because we have a STOP statement and not an assignment statement and therefore T5 cannot be used. Introducing new paths can only be done by T11, so this program cannot expand.

Theorem 9 showed that the complete set of transformations found by Theorem 8, is also irredundant.

5. OPTIMIZATION

We would like to provide a scheme for optimization in which a sequence of the transformations is applied for getting an optimal code.

DEFINITION

We say that a cost function on programs is reasonable provided that (1) the cost of a program decreases if statements which are never executed are deleted from the program in such a way that statements are not added to the program.

(2) the cost of a program decreases if a statement is deleted from some executable sequence of the program.

(3) the cost of a program decreases if identical subgraphs and identical statements can be merged in such a way that test statements are not added to the program.

If we consider a cost function which is some combination of speed and size of programs then
(1) reduces the size of the program. When the graph of the program is a tree, eliminating nonexecutable paths always reduces the size of the program. When the graph is not a tree- T11 reverse should be used in some cases to copy subgraphs so that nonexecutable paths may be eliminated (see Theorem 2) although sometimes the subgraphs can be merged again after nonexecutable paths have been eliminated by another application of T11. (1) includes also removing unreferenced blocks which always reduces the size of the program. (2) increases the speed of the program and decreases the size in some cases, although the size in other cases might be increased. The size of the program might increase when its graph is a tree, if T5 is used. When the graph is not a tree- T11 must

be used in some cases to copy subgraphs so that statements can be deleted (see Theorem 6). As in (1) in some cases subgraphs can be merged again after statements have been deleted. (3) reduces the size of the program.

We may conclude that we consider a combination of speed and size as reasonable, in which the size may be increased in order that the program will run faster, but otherwise the size is decreased whenever possible.

Other criteria of cost than speed and size can be introduced provided they do not conflict with (1)-(3) above. For example reducing the frequency of storing and recovering the partial results in computing arithmetic expressions is a reasonable cost criterion.

An optimal program for a given program Π, will be a program equivalent to Π whose cost can not be decreased.

Every optimal program is reduced (by (1) and (2) above) and if the graph of the program is a tree- each path of an optimal program is executable (by (1)).

The following theorem will give a general procedure for optimizing programs with no loops under reasonable cost functions, assuming the cost function is independent of the costs of conditional tests in the program. Reducing the number of conditional tests in the program can be done by reordering the tests. Reordering the tests also might minimize the average time through them when more information is given on the costs of different tests and the probabilities of their outcome. This type of optimization involves using T6, T7, T11 to add new tests and nonexecutable paths to the program and to eliminate other tests that become useless and paths that become nonexecutable. There is some difficulty in finding an efficient procedure for implementing this process using the transformations. Also if different sequences of tests result in executing different sequences

of assignment statements in the program, reordering of tests will not reduce the number of tests in the program. Therefore generally this type of optimization does not improve the code generated.

Although the optimization procedure of Theorem 10 does not minimize the number of tests in the program, it merges conditional tests appearing in identical subgraphs and it eliminates conditional branches to nonexecutable paths.

THEOREM 10

Assuming the cost function is independent of the costs of conditional tests in the program, there is an algorithm that finds an optimal program Π' equivalent to a given program Π. The algorithm operates in a series of steps so that Π is first transformed to an equivalent reduced program Π_c which is independent of the specific cost function used. Using T8 repeatedly Π_c is nondeterministically transformed to a program Π''. Π' is obtained by operating {T1,T3,T4,T5,T7,T11} on Π''.

PROOF:

We first assume that all nonexecutable paths can be eliminated from Π in such a way that statements are not added to the program. Thus in every optimal program Π' each path is executable. Also every optimal program is reduced. So take Π_1 to be a reduced program equivalent to Π in which each path is executable and no two statements in a path define the same variable. We can obtain Π_1 by first eliminating nonexecutable paths by T6 and T11 (Th.3) then reducing the obtained program by operating T1,T2,T4,T5,T9,T11 (Th.6) and then renaming variables by T3 such that no two statements in a path of Π_1 define the same variable. Thus

$$\Pi \xrightarrow[\{1,2,3,4,5,6,9,11\}]{*} \Pi_1.$$

Π_1 and Π' are reduced programs in which each path is executable and $\Pi_1 \equiv \Pi$, $\Pi' \equiv \Pi$, therefore $\Pi_1 \equiv \Pi'$. Using Th.4 we can find two programs Ψ and Ψ', $\Pi_1 \xrightarrow[\{1,5,7,8,11\}]{*} \Psi$, $\Pi' \xrightarrow[\{1,5,7,8,11\}]{*} \Psi'$ such that corresponding paths in Ψ and Ψ' are consistent and have the same tests. $\Pi_1 \equiv \Pi'$ therefore $\Psi \equiv \Psi'$. By looking at the algorithm of Th. 4 we observe that if Π_1 and Π' are reduced programs Ψ and Ψ' are also reduced. This is because the algorithm does not add useless statements and unreferenced blocks and also statements can be added to the programs in such a way that there are no redundant statements. By Th. 7 for all $i, D_i(\Psi) = D_i(\Psi')$ and by Th.5 $\Psi \xrightarrow[\{3,4,5,10,11\}]{*} \Psi'$. $\Pi_1 \xrightarrow[\{1,5,7,8,11\}]{*} \Psi \xrightarrow[\{3,4,5,10,11\}]{*} \Psi' \xrightarrow[\{1,5,7,8,11\}]{*} \Pi'$ thus $\Pi_1 \xrightarrow[\{1,3,4,5,7,8,10,11\}]{*} \Pi'$.

Because removing of unconditional useless test statements might expose pairs of tests that can be flipped by T8, we must operate T7 in the forward direction before operating T8. Starting with the program Π_1 we apply T10 to flip blocks and place after each unconditional test $t(A_1,\ldots,A_r)k,k$ in which k references the first statement of a block, the block referenced by k. This allows us to use T7 to remove all unnecessary unconditional tests. Next we apply T1 to remove any assignment statement which define variables that are referenced only by the unconditional tests just removed. The resulting program is called Π_c. $\Pi_1 \xrightarrow{\{1,7,10\}} \Pi_c$ and $\Pi_c \xrightarrow[\{1,3,4,5,7,8,10,11\}]{*} \Pi'$.

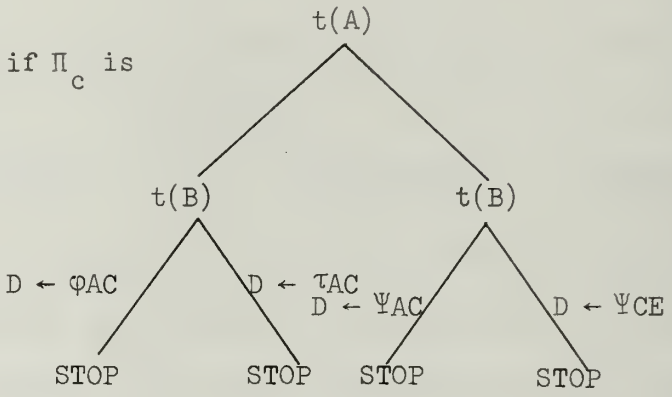
All the steps in going from Π to Π_c were deterministic. Π_c is independent of the given cost function.

Because flipping of tests might expose identical subgraphs and also might make merging of identical statements possible, we will first operate T8. We will operate T8 as many times as necessary so that

identical statements can be merged later, and also that identical subgraphs can be merged in such a way that the cost of the program is not increased. We obtain a program Π'' such that $\Pi_c \xRightarrow{\{8\}} \Pi''$.

Because applying T8 does not change the number of statements of the program, Π'' can be found in a finite number of steps, although a heuristic procedure might help to find Π'' quickly. Also different cost functions will cause different Π'' to be found. So an algorithm (or heuristic) based on the nature of the specific cost function might be useful.

e.g. if Π_c is



the statement D ← ΦAC might be translated into the following machine instructions:

a → acc

acc + c → acc

acc → d

where a,c,d are the addresses of A,C,D respectively and the interpretation In (ΦAC) = +AC. t(A) might be translated into:

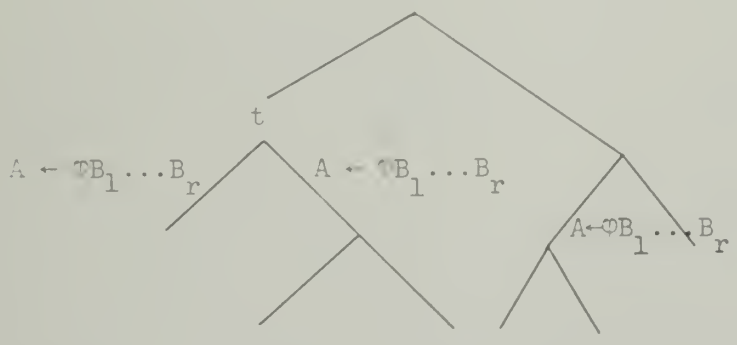
a → acc

t(acc)

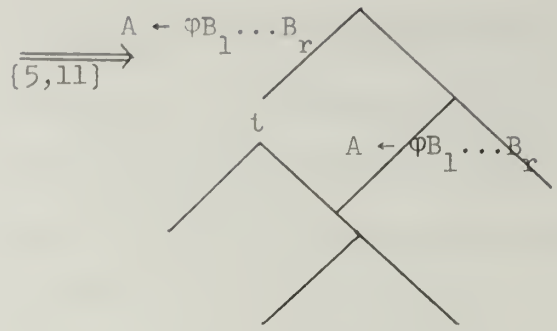
thus if we flip the tests t(A) and t(B) we can save one of the instructions a → acc. Therefore flipping of tests may improve the machine code obtained for some given cost functions.

Because $\Pi_c \xrightarrow[\{1,3,4,5,7,8,10,11\}]{*} \Pi'$ and we used T8 to obtain Π'' it follows that $\Pi'' \xrightarrow[\{1,3,4,5,7,10,11\}]{*} \Pi'$. All applications of T8 can be performed before applying {T1,T3,T4,T5,T7,T10,T11} . That is because T3,T4,T10,T11 and T5 in the forward direction do not expose new tests that can be flipped and all operations of T7 and T1 have been performed except those resulting from merging of paths.

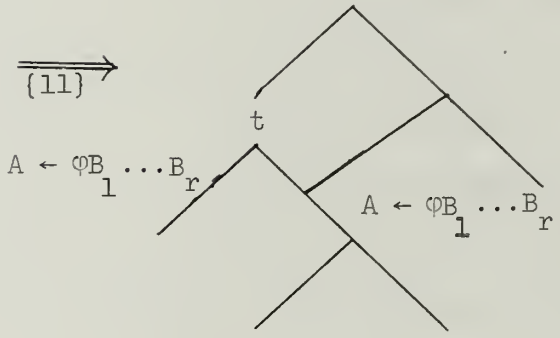
It is easy to see that T5 and T11 can be operated independently on Π'' in the sense that commuting the transformations will result in programs having the same cost. That is becuase if the subgraphs are identical and the two identical statements to be merged are included in both subgraphs, the resulting program is the same if we first merge the identical statements and later merge the identical subgraphs, or if we first merge the identical subgraphs and then merge the identical statements. If only one statement of the two identical statements to be merged is included in the identical subgraphs:



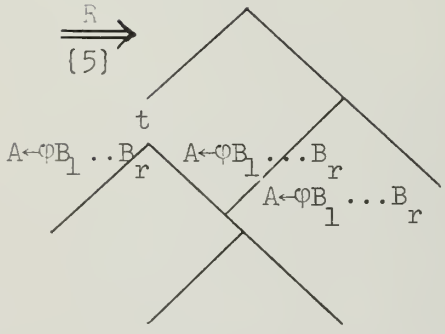
if we first merge the identical statements and then the identical subgraphs we get

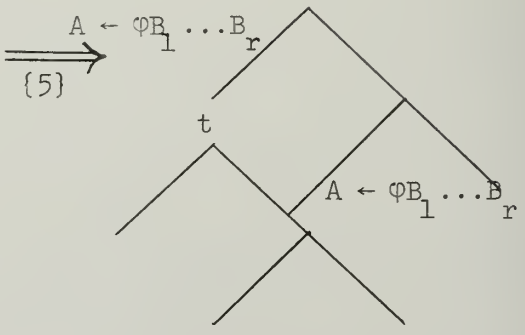


and if we first merge the subgraphs we get



which has the same cost as the program above. By applying T5 twice we can get the same result as above.





Thus we first operate T5 on Π" as many times as necessary to merge identical statements (we may have to use T3 and T4 to be able to do it) and then we use T11 to merge identical subgraphs. As was shown

above further applications of T5 might be used to get other programs with the same cost. T11 will be operated only in those cases in which identical subgraphs can be merged in such a way that tests are not added to the program.

Merging identical subgraphs might expose useless unconditional tests, so T7 has to be applied again. Again useless assignment statements can be removed by T1. We get a program we shall call Π_2. We have so far used T1,T3,T4,T5,T7 and T11 to transform Π'' into Π_2. Next we have to use T10 and again T3 and T4 to obtain the optimal program Π'.

T10 (flipping of blocks) does not change the cost of the program, it even does not change the graph of the program. So we might conclude that for a program Π, there might be several optimal programs all having the same cost and all having the same graph and all are equivalent by T10 only. Let Π' be one of them. Further applications of T10 to Π' may be used to generate other optimal programs.

Π' is obtained from Π_2 by applying T3 and T4.

The following part of the proof will be based on the arguments of Ullman and Aho (1) for the straight line code case which apply also here.

Since in Π_2 no variable is defined twice, the steps used in going from Π_2 to Π' can be reordered so that all applications of T4 are performed before applying T3. So the program Π_3 is a reordering of independent statements of Π_2, $\Pi_2 \overset{(4)}{\Longrightarrow} \Pi_3$ and Π' is renaming of the variables of Π_3, $\Pi_3 \overset{(3)}{\Longrightarrow} \Pi'$.

The step in going from Π_2 to Π_3 is nondeterministic. There are only finitely many possibilities for Π_3 because the number of statements

is not changed, but to avoid an exhaustive search for the optimal Π_3 some algorithm (or heuristics) based on the nature of the specific cost function can be used to get Π_3 quicker.

Since renaming does not change the cost- Π' could be taken to be equal to Π_3, and further applications of T3 may again be used to generate other optimal programs.

In the first part of the proof of Theorem 10, we assumed that nonexecutable paths can be eliminated so that statements are not added to the program. Therefore in the optimal program each path is executable.

Assume that not all nonexecutable paths of the program can be eliminated so that statements are not added to the program. Therefore take Π_1 to be a reduced program equivalent to Π so that no two statements in a path define the same variable. $\Pi \xrightarrow[\{1,2,3,4,5,9,11\}]{*} \Pi_1$.

$\Pi_1 \equiv \Pi$, $\Pi' \equiv \Pi$ therefore $\Pi_1 \equiv \Pi'$.

Let Ψ and Ψ' be two programs equivalent to Π_1 and Π' respectively, so that in Ψ and Ψ' each path is executable. By Theorem 3

$$\Pi_1 \xrightarrow[\{6,11\}]{} \Psi \qquad \Pi' \xrightarrow[\{6,11\}]{} \Psi'.$$

Let Ψ_1 and Ψ_1' be two programs equivalent to Ψ and Ψ' respectively so that corresponding paths in Ψ_1 and Ψ_1' are consistent and have the same tests. By Th. 4

$$\Psi \xrightarrow[\{1,5,7,8,11\}]{*} \Psi_1 \qquad \Psi' \xrightarrow[\{1,5,7,8,11\}]{*} \Psi_1'.$$

Ψ_1 and Ψ_1' are reduced, therefore by Th. 7 $D_i(\Psi_1) = D_i(\Psi_1')$ and by Th. 5

$$\Psi_1 \xrightarrow[\{3,4,5,10,11\}]{*} \Psi_1'.$$

Thus $$\Pi_1 \xrightarrow[\{1,3,4,5,6,7,8,10,11\}]{*} \Pi'.$$

Starting with the program Π_1 we operate T6 and T11 in all the cases that the number of statements added is not greater that the number

of statements eliminated. The rest of the algorithm is the same as in the first part of the proof of Theorem 10. We get a program Π_c, $\Pi_1 \xrightarrow{\{1,6,7,10,11\}} \Pi_c$. This completes the proof of Theorem 10.

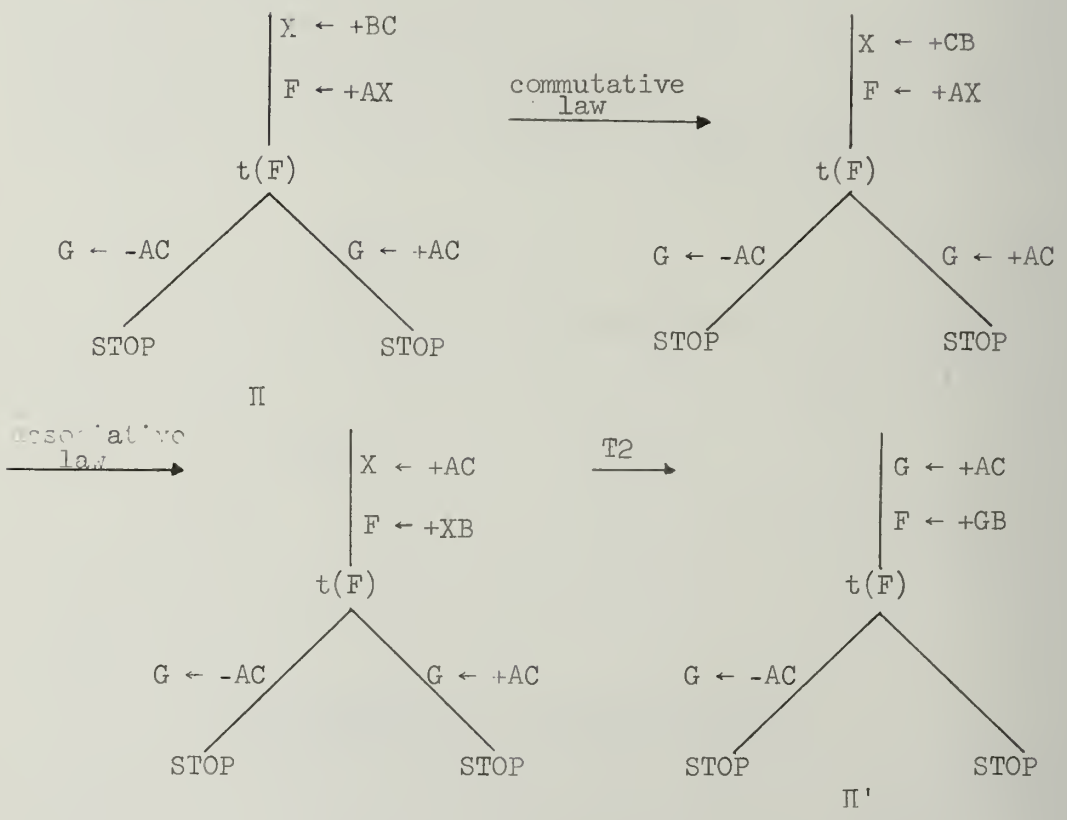
We presented a program schema that models loop-free programs. We found a complete set of equivalence preserving transformations and showed that this set is also irredundant. In this chapter we provided a procedure for optimization using the set of transformations. The results obtained will be extended to the case in which algebraic laws are assumed, and to the case in which the tests are Boolean combinations of elemental predicates. Also we will show that the results obtained hold for a broader class of programs- programs which always halt.

6. ALGEBRAIC TRANSFORMATIONS

Algebraic laws that are known to hold among operators can often be used to expose common subexpressions in the program. In the following example, the commutative and the associative laws of addition are used to expose a common subexpression which can later be eliminated, so that the cost of the program will be reduced.

EXAMPLE 14

Π = (G, {A,B,C} , {F,G})



The common subexpressions above could not be exposed using the topological transformations only.

In this chapter we use a model of program schemata in which assignment statements are of the form

$$A \leftarrow \Psi B_1 \ldots B_r$$

$A, B_1, \ldots, B_r \in \Sigma$ and Ψ is an r-ary operator from a countable set of operators Φ over the domain D.

To provide an actual program for a given program we assign values from the domain to the input variables and make a proper assignment of predicates to the test names of the schema.

The model assumes that a set of algebraic laws Γ holds among operators from Φ.

DEFINITION

An algebraic identity is a pair (α,β) where α, β are well-formed expressions over Φ and Σ.

Examples of algebraic identities are the associative law of addition (+A+BC, ++ABC) and the distributive law (*A+BC,+*AB*AC)

DEFINITION

Let E_1 and E_2 be two well-formed expressions, and $\gamma = (\alpha, \beta)$ is an algebraic identity. E_1 is transformed to E_2 by γ iff for each $x \in \Sigma$ appearing in α or β there is a well-formed expression E_x such that if α' and β' are α and β with E_x substituted for each instance of x, then either $E_1 = \delta_1 \alpha' \delta_2$ and $E_2 = \delta_1 \beta' \delta_2$ or $E_1 = \delta_1 \beta' \delta_2$ and $E_2 = \delta_1 \alpha' \delta_2$.

DEFINITION

Let E_1 and E_2 be two well-formed expressions and Γ is a set of algebraic laws. E_1 and E_2 are equivalent under Γ, $(E_1 \underset{\Gamma}{\equiv} E_2)$ iff E_1 can be transformed into E_2 by a finite sequence of laws in Γ.

DEFINITION

Let Π and Π' be two programs. Let ℓ and k be two executable paths of Π and Π' respectively. ℓ and k are said to be consistent under Γ iff for all t

(i) t is in the sequences of statements that correspond to both ℓ and k, so that in path ℓ there is a statement S_i^{ℓ} of the form $t^{\ell}(C_1,\ldots,C_r)\ n_1,n_2$, and in path k there is a statement S_j^k of the form $t^k(D_1,\ldots,D_r)\ m_1,m_2$ and

(ii) $v_i^{\ell}(C_q) \underset{\Gamma}{\equiv} v_j^k(D_q)$ for all q, $1 \leq q \leq r$, that is the expressions $v_i^{\ell}(C_q)$ and $v_j^k(D_q)$ are equivalent under Γ for all q, then the statement prefixed by n_1 is included in path ℓ iff the statement prefixed by m_1 is included in path k, and the statement prefixed by n_2 is included in path ℓ iff the statement prefixed by m_2 is included in path k.

For an aribtary set of algebraic identities it is recursively unsolvable to determine given two expressions E_1 and E_2 whether $E_1 \underset{\Gamma}{\equiv} E_2$, thus it is undecidable if two paths are consistent.

In Example 14 above the two left paths ℓ_1 and k_1 of Π and Π' respectively are consistent under a set of algebraic identities Γ that includes the associative and the commutative laws of addition.

$v^{\ell_1}(F) = +A+BC$

$v^{k_1}(F) = ++ACB$

$+A+BC \underset{\Gamma}{\equiv} ++ACB$

DEFINITION

Two program schemata Π and Π' will be called equivalent under a set of algebraic laws Γ ($\Pi \underset{\Gamma}{\equiv} \Pi'$) iff for all consistent pairs of paths under Γ ℓ,k, for each expression E^{ℓ} in $v^{\ell}(\Pi)$ there is an expression E^k in $v^k(\Pi')$ such that $E^{\ell} \underset{\Gamma}{\equiv} E^k$, and conversely.

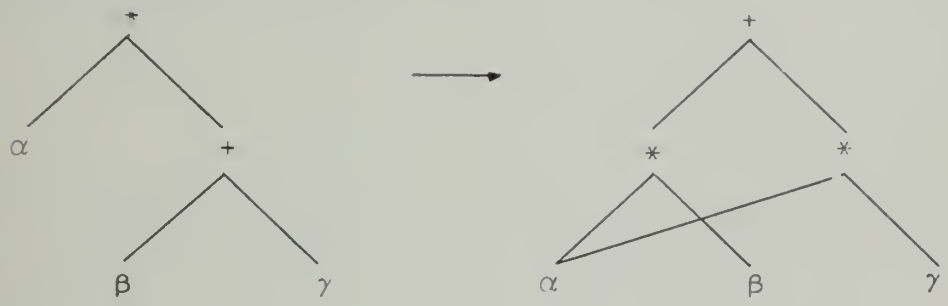
Because it is undecidable for an arbitary Γ whether two expressions are equivalent under Γ, the question of equivalence under arbitary Γ for two programs Π and Π' is undecidable.

Algebraic laws on expressions induce transformations on dags. Thus algebraic transformations on programs can be described as transformations on the corresponding dags.

If the distributive law holds for expressions α, β, γ

$$*\alpha + \beta\gamma = + *\alpha\beta *\alpha\gamma$$

then the transformation on the dag is



D_1 D_2

The transformation on dags induce a natural transformation on any given dag of the program: we replace each instance of D_1 with D_2.

Before replacing instances of dags that correspond to algebraic identities in a dag of a given program, we must make sure that

1) No node of the dag replaced (which is not a root) is distinquished. (otherwise distinguished nodes associated with output variables and with variables referenced by tests might be eliminated by the transformation).

2) No node of the dag replaced (which is not the root or a leaf) has ancestors which are not nodes of the dag replaced in all the dags that include the node. (because nodes might be eliminated or their values might be changed by an algebraic transformation).

3) No node is not distinguished and has no ancestors (because then the node is useless, and there is no use in transformations on useless statements).

4) T11 has been used so that there is no path that includes a subexpression of the expression replaced, but not the expression itself.

DEFINITION

We will call an open program a program in which

a) no output variable is referenced after it is defined.

b) no variable not in I, is referenced more than once (in all paths that include the definition of this variable).

c) T1 can not be applied.

d) T11 reverse can not be applied.

e) no two output variables have the same value.

a-d will guarantee 1-4 above.

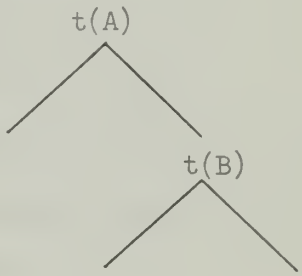
By Lemma 8 below, each program can be transformed to an equivalent open program. Thus if I is an algebraic identity and Π and Π' are two programs in which corresponding paths are consistent under I we say

$$\Pi \underset{I}{\Longrightarrow} \Pi' \qquad \text{iff} \qquad D_i(\Pi_0) \underset{I}{\Longrightarrow} D_i(\Pi_0')$$

where Π_0 and Π_0' are open programs equivalent to Π and Π' respectively, and D_i are all the dags in $D(\Pi_0)$ that include the subtree corresponding to the expression transformed by I.

Because by Theorem 5 dags characterize equivalence classes under the transformations T3,T4,T5,T10,T11, the definition above allows I to incorporate all these transformations.

We will be interested in algebraic transformations that operate on programs in which each path is executable, because there is no use to expose common subexpressions in nonexecutable paths. For an arbitrary set of identities Γ, it is undecidable if a path is executable if algebraic laws are assumed to hold among operators. For special cases of algebraic identities getting rid of nonexecutable paths can be done by a similar procedure to that of Theorem 3. We apply a transformation similar to T6 in which the left path is removed if $v(A) \underset{\Gamma}{\equiv} v(B)$.



Lemma 8

If Π is a program in which each path is executable, there is an open program Π_0 such that $\Pi \equiv \Pi_0$ and $\Pi \overset{*}{\underset{\{1,2,9,11\}}{\Longrightarrow}} \Pi_0$.

PROOF:

First we reduce Π so that useless statements will be eliminated and no two output variables will have the same value. We get a program Π_1, $\Pi \overset{*}{\underset{\{1,2,9,11\}}{\Longrightarrow}} \Pi_1$. Now we operate T11 reverse as many times as necessary so that we get a tree. We rename Π_1 such that no variable will be defined more that once in the same path (T3). We get Π_2, $\Pi_1 \overset{*}{\underset{\{3,11\}}{\Longrightarrow}} \Pi_2$ and by Lemma 1, $\Pi_1 \overset{*}{\underset{\{1,2,11\}}{\Longrightarrow}} \Pi_2$. Now we use T2 reversed (inserting redundant statements) as many times as necessary so that no output variable will be referenced after it is defined, and also that no variable will be referenced more than once. We get an open program Π_0, $\Pi_2 \overset{*}{\underset{\{2\}}{\Longrightarrow}} \Pi_0$ and $\Pi \overset{*}{\underset{\{1,2,9,11\}}{\Longrightarrow}} \Pi_0$.

We will prove an equivalent theorem to Th. 8 for the case that algebraic identities are assumed:

THEOREM 11

Let Π and Π' be two programs in which each path is executable. Let Γ be a set of algebraic identities. Then

$$\Pi \underset{\Gamma}{\equiv} \Pi' \quad \text{iff} \quad \Pi \xrightarrow[\{1,2,6,7,9,10,11\} \cup \Gamma]{*} \Pi'.$$

When algebraic identities are assumed to hold among operators, the topological transformations that operate on tests consider identical tests as tests checking values that are equivalent under Γ. When equivalence under Γ of two programs is decidable, it is decidable whether the values checked by tests appearing in the program are equivalent under Γ.

If Π and Π' are two programs in which each path is executable and $\Pi \underset{\Gamma}{\equiv} \Pi'$, then $\Pi \xrightarrow[\{1,5,7,8,11\}]{*} \Psi$, $\Pi' \xrightarrow[\{1,5,7,8,11\}]{*} \Psi'$ such that in Ψ and Ψ' corresponding paths are consistent under Γ. The procedure is the same as in Theorem 4, only here tests that check equivalent expressions under Γ are considered to be identical.

PROOF:

1) "if" is trivial because the transformations preserve equivalence.

2) Let Ψ and Ψ' be two programs equivalent to Π and Π' respectively, in which corresponding paths are consistent under Γ.

$$\Pi \xrightarrow[\{1,5,7,8,11\}]{*} \Psi \qquad \Pi' \xrightarrow[\{1,5,7,8,11\}]{*} \Psi'$$

and by Lemmas 3 and 4 $\quad \Pi \xrightarrow[\{1.2,6,7,11\}]{*} \Psi \quad \Pi' \xrightarrow[\{1,2,6,7,11\}]{*} \Psi'.$

$$\Pi \underset{\Gamma}{\equiv} \Pi' \quad \text{therefore} \quad \Psi \underset{\Gamma}{\equiv} \Psi'.$$

Let Π_0 and Π_0' be open programs for Ψ and Ψ' respectively.

$$\Psi \xrightarrow[\{1,2,9,11\}]{*} \Pi_0 \qquad \Psi' \xrightarrow[\{1,2,9,11\}]{*} \Pi_0', \quad \Pi_0 \equiv \Pi_0'.$$

$$\text{Let } v(\Pi_0) = \left\{ v^{\ell_i}(\Pi_0) \right\}_{i=1}^{n} = \left\{ E_1^{\ell_i}, \ldots, E_q^{\ell_i} \right\}_{i=1}^{n}$$

There is a finite sequence of algebraic identities that operate on each set $\{E_j^{\ell_i}\}_{i=1}^{n}$ $1 \leq j \leq q$ and convert it to the set of expressions $\{E'^{k_i}_j\}_{i=1}^{n}$ where $\{\{E'^{k_i}_j\}_{i=1}^{n}\}_{j=1}^{q} = v(\Pi_0')$. So there is a sequence of program values $v_1, \ldots, v_r$ such that $v_1 = v(\Pi_0)$ $v_r = v(\Pi_0')$ and v_{i+1} is created from v_i by applying one identity in Γ to one or more expressions in v_{i+1}. An identity operates simultaneously on several expressions that are the values of the same output variable in all the paths of the program.

We will show that if ρ is an open program and $v(\rho) = v_i$, then there is a program ρ' such that $v(\rho') = v_{i+1}$ and $\rho \xrightarrow[\{1,2,6,7,10,11\} \cup \Gamma]{} \rho'$.

Let I be the identity applied in going from v_i to v_{i+1}. Since ρ is open, I is applicable to the dags in $D(\rho)$. After operating on all the dags $D_i(\rho)$ that include the subtree corresponding to the expression transformed by I, we get a new set of dags D_1 whose value is v_{i+1}. Let ρ' be such that $D(\rho') = D_1$. Thus $\rho \xRightarrow{\Gamma} \rho'$. Also because by Theorem 5 dags characterize equivalence classes under T3,T4,T5,T10,T11 $\rho \xrightarrow[\Gamma \cup \{1,2,10,11\}]{*} \rho'$. (We used Lemmas 1,2,3).

Therefore $\Pi \xrightarrow[\{1,2,6,7,9,10,11\} \cup \Gamma]{} \Pi'$.

The following theorem will give a procedure for optimizing programs under operator and operand preserving algebraic identities.

DEFINITION

An algebraic identity is operator preserving if the number of operators on both sides of the identity is the same.

The identity --ABC- -A+BC is operator preserving. Under an operator preserving identity the number of assignment statements is preserved.

DEFINITION

An algebraic identity is operand preserving if each operand either

i) appears exactly once on each side of the identity, or

ii) all its appearances on a side of the identity follow the same instance of an operator.

Under an operand preserving identity, openness of programs is preserved.

The definition of operand preserving identity includes that of Aho and Ullman (1).

**XXY = *Y*XX is an operand and operator preserving identity.

THEOREM 12

Let Γ be a set of operator and operand preserving identities.

There is an algorithm that finds for every program Π an optimal program Π' such that $\Pi \underset{\Gamma}{\equiv} \Pi'$. The algorithm operates in a series of steps so that Π is first transformed to an equivalent open program Π_0. Using algebraic transformations only, Π_0 is then nondeterministically transformed to Π_0'. Π' is obtained by operating the topological transformations {T1,T2,T3,T4,T5,T7,T8,T10,T11} on Π_0'.

PROOF:

Let Π_0 be an open program equivalent to Π in which each path is executable. Π_0 can be obtained by first eliminating nonexecutable paths

by a similar procedure to that of Theorem 3, and then costructing an open program by using T1,T2,T9,T11 (Lemma 8).

Since Π_0 is open and the algebraic identities in Γ preserve openness, the algebraic transformations do not affect the relative position of branch nodes of the dags that are not replaced by the algebraic transformations. Thus all the algebraic transformations can be applied before applying other transformations that operate on assignment statements. It is also clear that algebraic transformations can be applied before applying transformations that operate on tests, because they change only assignment statements. Thus we can transform Π_0 by applying algebraic transformations only, into a program Π_0' such that $\Pi_0' \equiv \Pi'$.

Operator and operand preserving identities preserve openness, so that the algebraic transformations can be applied sequentially, without adding statements for making the obtained programs open. Because the number of statements is preserved, Π_0' can be found in a finite number of steps, although a heuristic procedure might help to find Π_0' quicker. Now we reduce Π_0' and apply T5 to remove variables, such that no two statements in a path will define the same variable. We obtain a program Π_1. The rest of the proof is exactly as in Theorem 10.

There are two generally recognized ways to optimize code generated by a compiler: global optimization which is concerned with the whole program, and local optimization which depends only on information in a single expression or statement. Theorems 10 and 12 provided schemes for global optimization without algebraic identities, and with certain types of identities, respectively.

Some of the local optimization techniques presented in literature Baywell (3) Lowry and Medlock (7) could be viewed as algebraic identities that hold among operators, operands and constants. Examples of algebraic identities that induce local optimization of code are

$$X * 2 = X + X$$

$$X ** 2 = X * X$$

$$X ** \frac{1}{2} = SQRT\ (X)$$

$$A ** (-C) = \frac{1}{A**C}$$

$$A ** 2. = A ** 2$$

$$X * 1 = X$$

$$X + 0 = X$$

In these cases the object of optimizations under algebraic identities is not to expose common subexpressions as before, but to locally improve the machine code generated. Improvements are usually done by replacing some operators with operators that are known to be more efficient in these special cases.

In the case of global optimization, a cost was reasonable provided that the cost decreased if a statement was deleted from some executable sequence of the program. In the cases of local optimization, a cost criterion is called reasonable provided that the cost decreases if a subsequence of statements in some executable sequence of the program is replaced by another subsequence of statements which is not longer and which runs faster.

We will extend the procedure in Theorem 12 to apply to certain types of algebraic identities that induce local optimization of code.

Because these identities operate on constants in addition to operands and operators, we will extend the definition of a program schema to include constants. The extension does not pose any difficulties and will be described informally.

Σ, Φ, T will be as before and C^t will be a countable set of constants.

assignment statements will be of the type

k $\quad A \leftarrow \theta B_1 \ldots B_r$

$A \in \Sigma$. $B_1, \ldots, B_r \in \Sigma \cup C^t$ and $\exists i$ $1 \leq i \leq r$, $B_i \in \Sigma$

test statements will be of the type

k $\quad t(C_1, \ldots, C_r)\ k_1, k_2$

$C_1, \ldots, C_r \in \Sigma \cup C^t$ and $\exists i$ $1 \leq i \leq r$, $C_i \in \Sigma$

A program schema will be a quadruple (P, I, U, C) where P, I, U are as before, and C is a finite set of constants.

Program equivalence will be defined as before.

The graphical representation of programs as dags will be as before, and a leaf will be created for each $c \in C$ in all the dags of the program.

The definitions of operand and operator preserving identities will be extended to include constants. The restrictions on operators and operands are as before but constants can appear anywhere on the two sides of the identity. By the new definition the identities

$X * 2 = X + X$

$X ** 2 = X * X$

$X ** \frac{1}{2} = SQRT(X)$

$$A \text{ ** } (-C) = \frac{1}{A \text{ ** } C}$$

$$A \text{ ** } 2. = A \text{ ** } 2$$

are all operand and operator preserving.

$$A + X * 1 = A + X \quad \text{and}$$

$$A + X + 0 = A + X$$

are not, because they are not operator preserving.

Under the new definition the number of statements is preserved, and also the openness of the programs is preserved because constants appear as leaves in the dags of the program. Therefore the scheme for optimization provided by Theorem 12 applies also to the extended case of operator and operand preserving algebraic identities. The cost function is reasonable also in the sense that the cost decreases if local improvements are made. Thus Theorem 12 provides a scheme for global and local optimization where certain types of algebraic identities are assumed.

7. LOGICAL TRANSFORMATIONS

We considered a program schema that contained a countable set T of test names. The elements of T will be called elementary tests.

DEFINITION

A test is either an

(1) elementary test, or

(2) an expression of the form $p \vee r$ where p, r are tests, or

(3) an expression of the form $p.r$ where p, r are tests, or

(4) an expression of the form $\bar{p}$ where p is a test.

The tests with the binary operations $\vee$ and . and with the set B consisting of the two constants T, F can be shown to be the algebra of Boolean functions of the elementary tests.

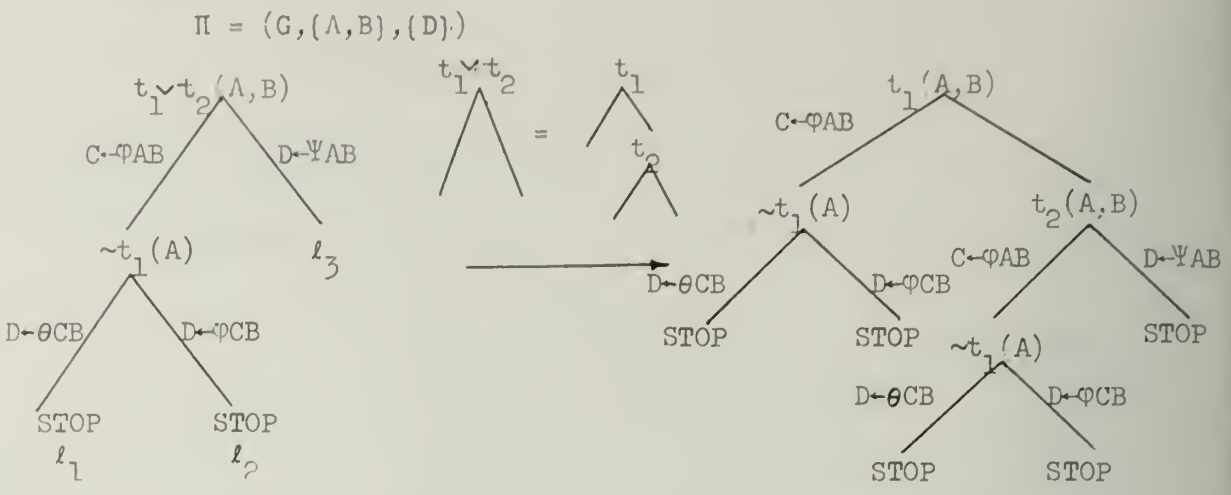
The definition of a program schema is extended to include test statements that have both elementary tests and tests.

An interpretation is defined as before.

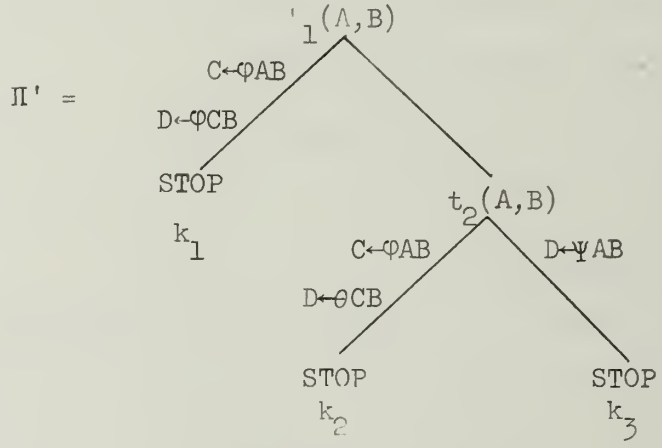
The definitions of executable and nonexecutable paths are as in the case of program schemata that have elementary tests only.

Boolean manipulations on the tests may reduce the cost of the program. In the following example we apply Boolean manipulations to expose nonexecutable paths that can later be eliminated so that the cost of the program is reduced.

EXAMPLE 15



The path that includes the tests $t_1(A,B)$ and the left branch of $\sim t_1(A)$ and the path that includes $t_1(A,B)$, $t_2(A,B)$ and the right branch of $\sim t_1(A)$ are nonexecutable. By eliminating them we get

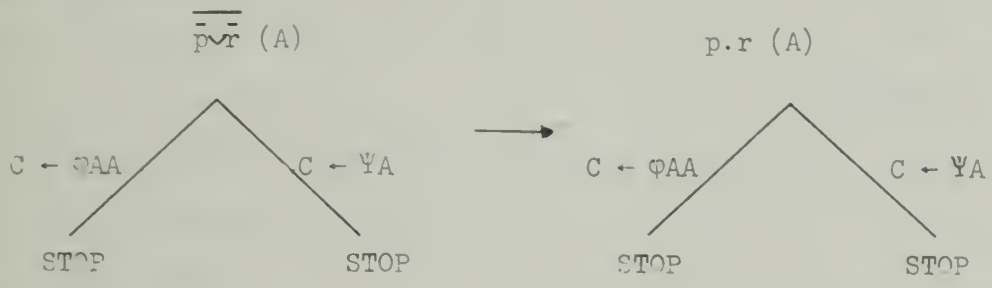


The logical transformations exposed nonexecutable paths of the program, also tests were eliminated from some executable sequences of the program. Thus logical transformations, in addition to topological and algebraic transformations, might reduce the cost of programs that have tests.

Optimizing compilers that use logical transformations have been built (see (5)). In FORTRAN, logical transformations may simplify the

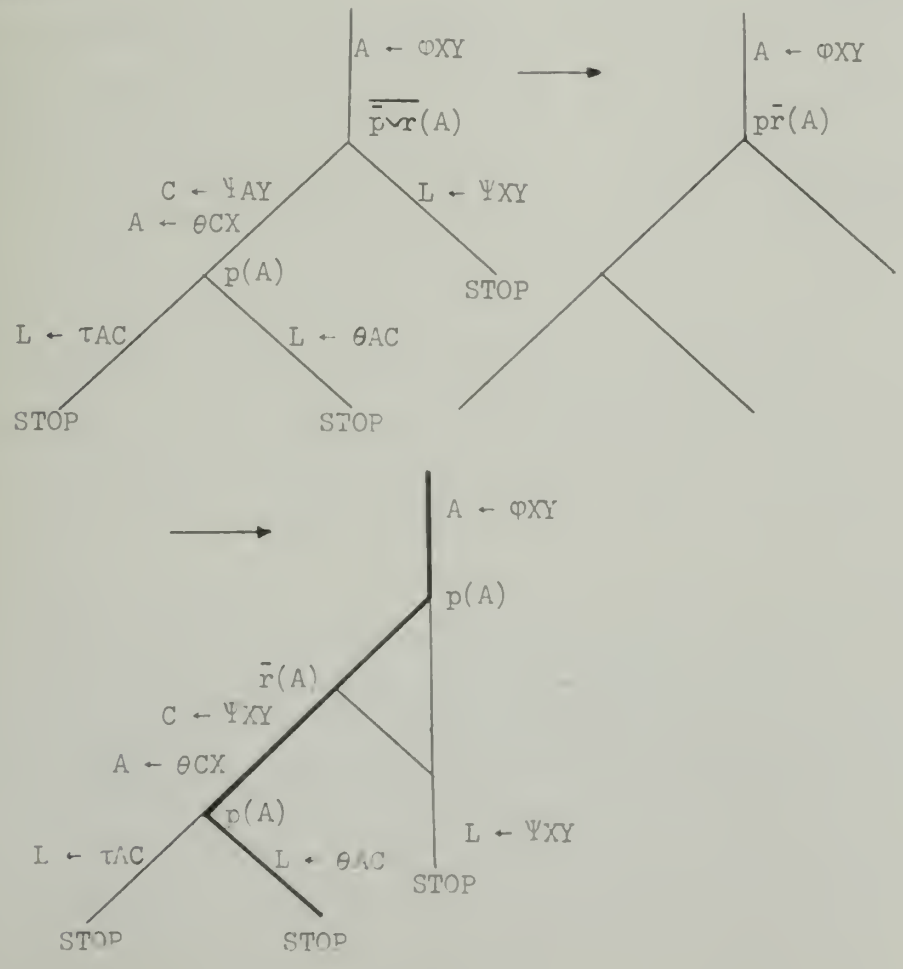
logical expressions in IF statements. See the reference above and also Lowry and Medlock (7).

The cost of the program can also be reduced by simplifying the Boolean expressions:



Two identical tests checking different values are considered to be different variables in our Boolean algebra:

EXAMPLE 16



The value of the variable A checked by the first p(A) test is ΦXY and by the second p(A) test is $\theta\Psi XYX$. Therefore the marked path can not be eliminated.

We will define the notion of equivalence of loop-free programs that have tests in addition to elementary tests.

DEFINITION

Let Π be a loop-free program that has tests. With each executable path ℓ_i of the program we will associate a path condition Ψ^{ℓ_i} which is the condition that path ℓ_i is executed, and is a Boolean function of the tests.

Identical tests checking different values are considered as different variables of the Boolean function.

DEFINITION

Let Π and Π' be two programs that have tests and ℓ and k are two executable paths of Π and Π' respectively. ℓ and k are said to be consistent iff Ψ^{ℓ} and Ψ^{k} are Boolean functions of the elementary tests $t_1, t_2, \ldots, t_m$, there is some assignment of truth values $a_1, \ldots, a_m$ to the variables $t_1, t_2, \ldots, t_m$ which makes both Ψ^{ℓ} and Ψ^{k} true.

Notice that the definition of consistent paths for programs that have elementary tests only, is a special case of this definition where Ψ^{ℓ} and Ψ^{k} are products of the elementary tests appearing in the paths.

DEFINITION

Let Π and Π' be two loop-free programs that have tests. Π and Π' will be called equivalent iff for all interpretations In (of input values, elementary tests and functions) $\mathrm{Val}_{\mathrm{In}}(\Pi) = \mathrm{Val}_{\mathrm{In}}(\Pi')$.

THEOREM 13

Let Π and Π' be two programs that have tests. $\Pi \equiv \Pi'$ iff for all consistent pairs of paths ℓ, k, $v^{\ell}(\Pi) = v^{k}(\Pi')$.

PROOF:

The proof is similar to that of Theorem 2.

For 1) we take the same interpretation In of inputs and functions with the following interpretation for elementary tests:

For all elementary tests t^j in path $j=\ell,k$

i) determine $In(t^j(\alpha_1,\ldots,\alpha_n))$ for $(\alpha_1,\ldots,\alpha_n)$ such that $\alpha_i = v_m^j(X_i)$ $1 \leq i \leq n$ and $t^j(X_1,\ldots,X_n)$ appears in the test statement S_m^j, in such a way that both Ψ^{ℓ} and Ψ^{k} are true.

ii) $In(t^j(\alpha_1,\ldots,\alpha_n))$ is arbitrary for other n-tuples $(\alpha_1,\ldots,\alpha_n)$.

ℓ and k are consistent therefore there is an assignment of the elementary tests which makes both Ψ^{ℓ} and Ψ^{k} true. Under this interpretation paths ℓ and k are executed in Π and Π' respectively.

The rest of 1) and also 2) are the same as in Theorem 2.

EXAMPLE 17

Let Π and Π' be of Example 15. The pairs of paths (ℓ_1,k_2), (ℓ_2,k_1), (ℓ_3,k_3) are consistent and their values are identical.

$\Psi^{\ell_1} = (t_1 \vee t_2)\,\bar{t}_1 = t_2\bar{t}_1$

$\Psi^{k_2} = \bar{t}_1 t_2$

$\Psi^{\ell_1} = \Psi^{k_2}$ thus there is an assignment of truth values to t_1 and t_2 that makes both Ψ^{ℓ_1} and Ψ^{k_2} true. Therefore ℓ_1 and k_2 are consistent.

$v^{\ell_1}(\Pi) = ABB = v^{k_2}(\Pi')$

$$\Psi^{\ell_2} = (t_1 \vee t_2)\,\bar{t}_1 = t_1 \vee t_1 t_2 = t_1$$

$$\Psi^{k_1} = t_1$$

thus $\Psi^{\ell_2} = \Psi^{k_1}$

$$v^{\ell_2}(\Pi) = \varphi\varphi ABB = v^{k_1}(\Pi')$$

$$\Psi^{\ell_3} = \overline{t_1 \vee t_2}$$

$$\Psi^{k_3} = \bar{t}_1 . \bar{t}_2$$

thus $\Psi^{\ell_3} = \Psi^{k_3}$.

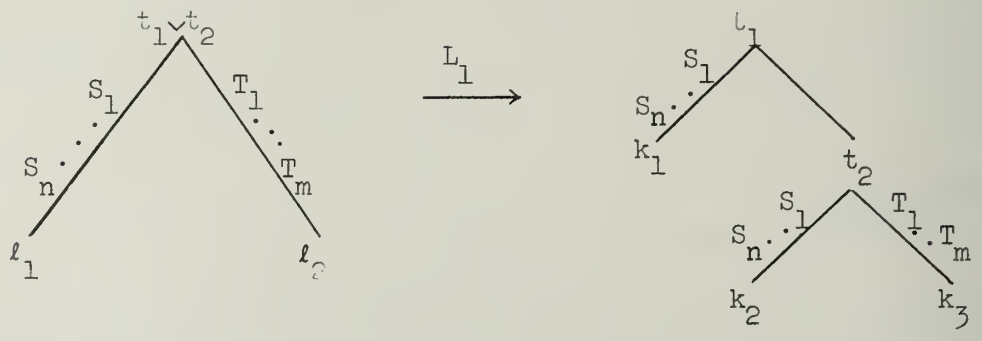
$$v^{\ell_3}(\Pi) = \Psi AB = v^{k_3}(\Pi')$$

In Π and Π' consistent paths have the same value, therefore by Theorem 13 $\Pi \equiv \Pi'$.

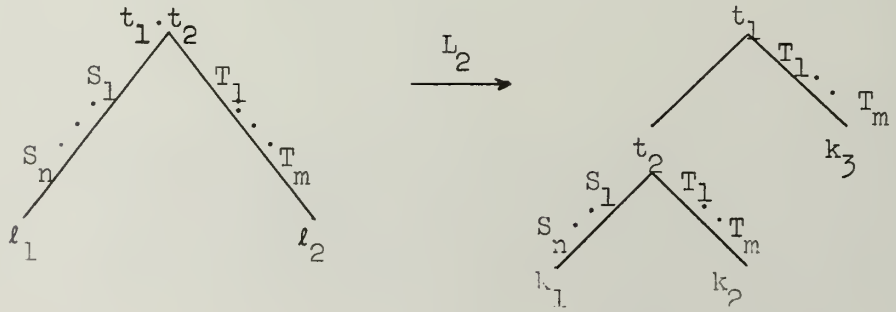
EXAMPLE 18

The following logical transformations preserve equivalence.

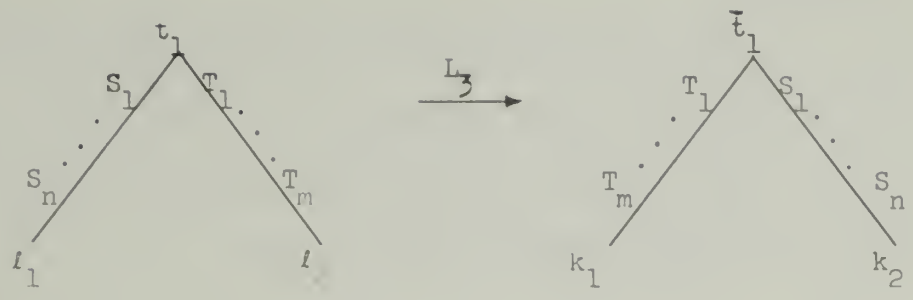
i)



ii)



iii)



PROOF:

In i) $\Psi^{\ell_1} = t_1 \vee t_2, \Psi^{k_1} = t_1$. There is an assignment of truth values to t_1 and t_2 which makes both Ψ^{ℓ_1} and Ψ^{k_1} true. (take t_1=T, t_2=F). Thus ℓ_1 and k_1 are consistent. The values of ℓ_1 and k_1 are identical.

$\Psi^{k_2} = \bar{t}_1 t_2$. The assignment t_1=F, t_2=T makes both Ψ^{ℓ_1} and Ψ^{k_2} true, therefore k_2 and ℓ_1 are consistent. The values of k_2 and ℓ_1 are identical.

$\Psi^{\ell_2} = \overline{t_1 \vee t_2} = \Psi^{k_3}$. Thus ℓ_2 and k_3 are consistent. Their values are identical.

Therefore L_1 preserves equivalence.

ii) $\Psi^{\ell_1} = t_1 t_2 = \Psi^{k_1}$. The values of ℓ_1 and k_1 are identical.

$\Psi^{\ell_2} = \overline{t_1 t_2}$ $\Psi^{k_2} = t_1 \bar{t}_2$

The assignment t_1=T, t_2=F makes both Ψ^{ℓ_2} and Ψ^{k_2} true, therefore ℓ_2 and k_2 are consistent. The values of ℓ_2 and k_2 are identical.

$\Psi^{k_3} = \bar{t}_1$

$\Psi^{\ell_2} = \bar{t}_1 \vee \bar{t}_2$

The assignment t_1=F, t_2=F makes both Ψ^{ℓ_2} and Ψ^{k_3} true, therefore ℓ_2 and k_3 are consistent. Their values are identical.

Therefore L_2 preserves equivalence.

iii) The pairs (ℓ_1, k_2), (ℓ_2, k_1) are consistent and their values are the same, therefore L_3 preserves equivalence.

DEFINITION

Any Boolean manipulation on the tests that preserve equivalence will be called a logical transformation on the program.

We will prove an equivalent theorem to Theorem 8, for the case that logical transformations are included:

THEOREM 14

Let Π and Π' be two loop-free programs that have tests. Then

$$\Pi \equiv \Pi' \quad \text{iff} \quad \Pi \xrightarrow[\{1,2,6,7,9,10,11\}\cup L]{*} \Pi'$$

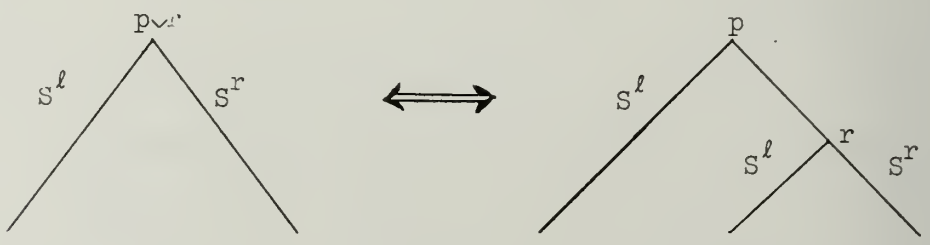
where L is a set of logical transformations.

PROOF:

1) "if" is trivial because logical transformations and topological transformations preserve equivalence of programs.

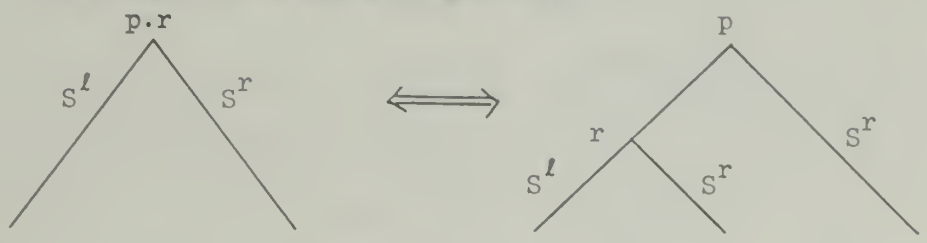
2) For every loop-free program with tests there is an equivalent loop-free program with elementary tests only, which is obtained by applying logical transformations. We will show that by induction on the structure of a test.

If a test is of the form $p \vee r$ then



where S^l and S^r are the left and right subtrees, respectively. (We assume that the graph of the given program is a tree. This assumption does not cause any difficulties since by repeated applications of T11 in reverse any program can be transformed to an equivalent program whose graph is a tree).

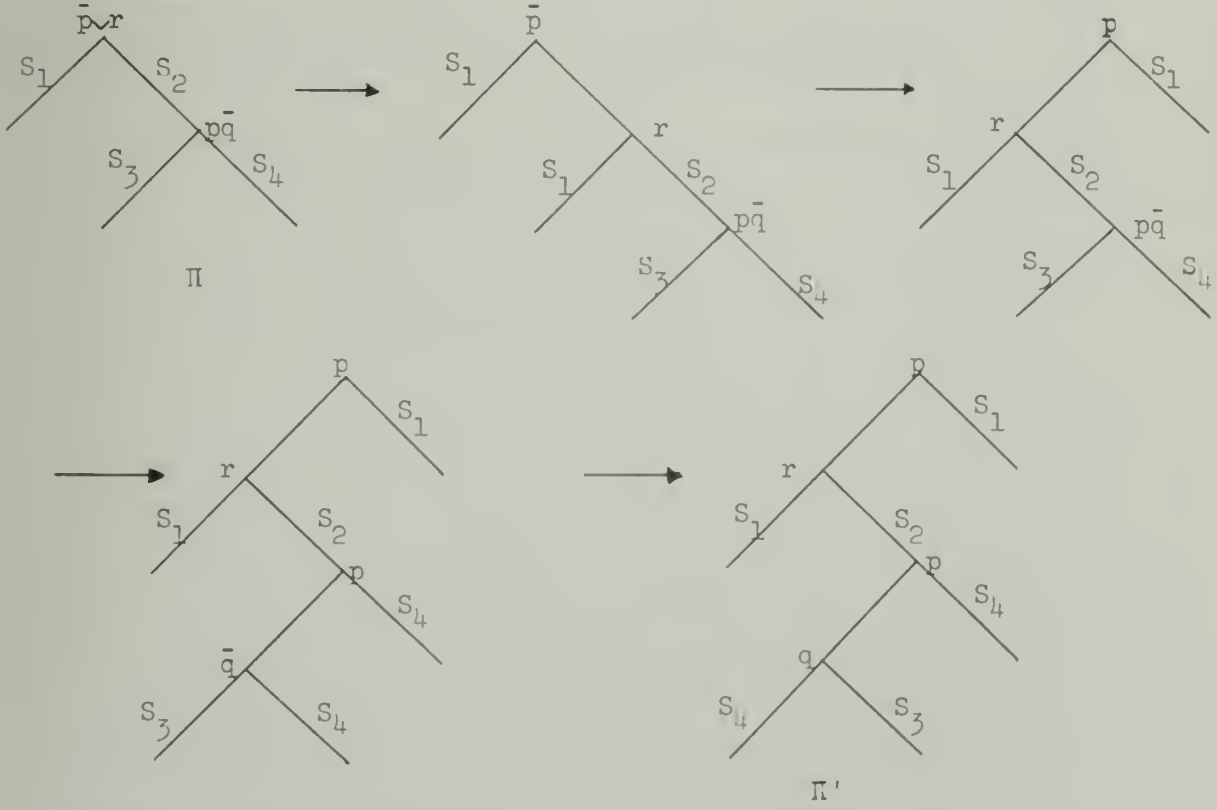
If a test is of the form p.r then



If a test is of the form $\bar{p}$ then

These transformations preserve equivalence (see Example 18).

EXAMPLE:



$\Pi \equiv \Pi'$ and Π' has elementary tests only.

Let Π_1 and Π_1' be loop-free programs that have elementary tests only, equivalent to Π and Π' respectively.

$$\Pi \xrightarrow[\{11\} \cup L_1]{} \Pi_1 \qquad \Pi \equiv \Pi_1 .$$

T11 is needed to transform the graph of Π to a tree.

$$\Pi' \xrightarrow[\{11\} \cup L_2]{} \Pi_1' \qquad \Pi' \equiv \Pi_1' .$$

L_1 and L_2 are all the logical transformations required to transform Π to Π_1, and Π' to Π_1', respectively. It follows that $\Pi_1 \equiv \Pi_1'$. Π_1 and Π_1' are loop-free programs with elementary tests only. Therefore by Theorem 8

$$\Pi_1 \xrightarrow{\{1,2,6,7,9,10,11\}} \Pi_1' .$$

Thus $$\Pi \xrightarrow[\{1,2,6,7,9,10,11\} \cup L_1 \cup L_2']{} \Pi'$$

where L_2' are all the logical transformations required to transform Π_1' to Π'. Let $L = L_1 \cup L_2'$. Thus

$$\Pi \xrightarrow[\{1,2,6,7,9,10,11\} \cup L]{} \Pi'$$ where L is a set of logical transformations.

8. OPTIMIZATION UNDER LOGICAL TRANSFORMATIONS

The object of optimization under logical transformations is

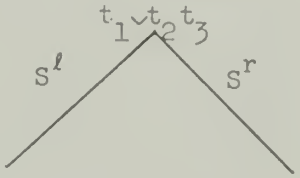
i) to eliminate nonexecutable paths.

ii) to minimize the number of tests performed.

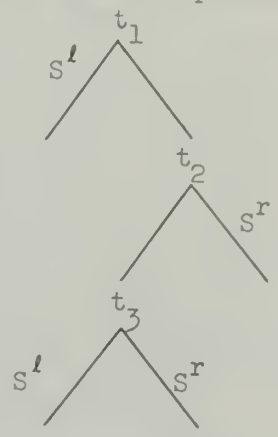
Example 15 above showed how logical transformations may reduce the cost of the program by eliminating nonexecutable paths. Minimizing the number of tests performed is done by (a) simplifying the Boolean expressions of the tests and (b) by breaking down the logical expressions so that only the necessary minimum number of tests are performed.

EXAMPLE 19

The program Π_1 has the following test where S^l and S^r are the left and right subtrees, respectively.



The logical expression is completely determined if t_1 is true, and there is no need in this case to perform the tests t_2 and t_3. Also, if t_1 is false and one of t_2, t_3 is false, there is no need to perform the other test. Thus by transforming Π_1 to Π_2



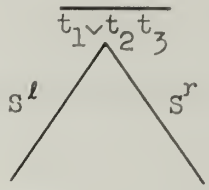
we reduce the number of tests performed in the following cases:

1) t_1 is true

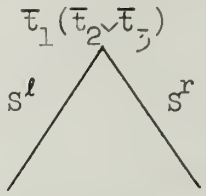
2) t_1 is false and t_2 is false.

In all the other cases, the cost of the program is unchanged.

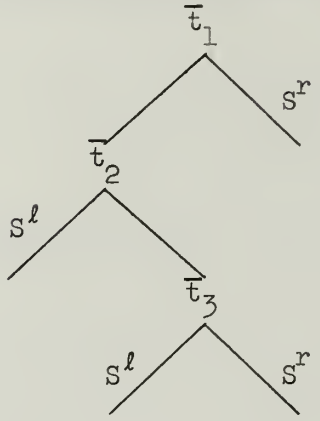
If Π_1 has the test -



we first have to simplify the Boolean expression by de Morgan's law



and now we apply the arguments above and break down the tests. We get



Compilers that break down logical expressions in order to accelerate the execution of logical statements have been constructed. See Lowry and Medlock (7) and Huskey and Wattenberg (4).

The definition of a reasonable cost for programs with elementary tests is changed to include (ii) above.

DEFINITION

A cost function on programs that have tests is reasonable provided that

1) the cost decreases if statements which are never executed are deleted from the program in such a way that statements are not added to the program.

2) the cost decreases if a statement is deleted from some executable sequence of the program.

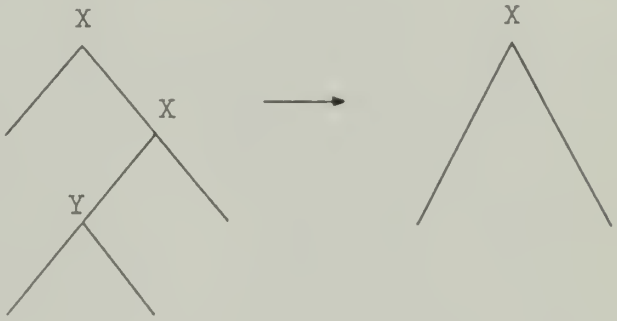
3) the cost decreases if identical subgraphs and identical statements are merged in such a way that tests are not added to the program.

4) the cost decreases if Boolean expressions of tests are broken down so that the number of elementary tests performed is reduced.

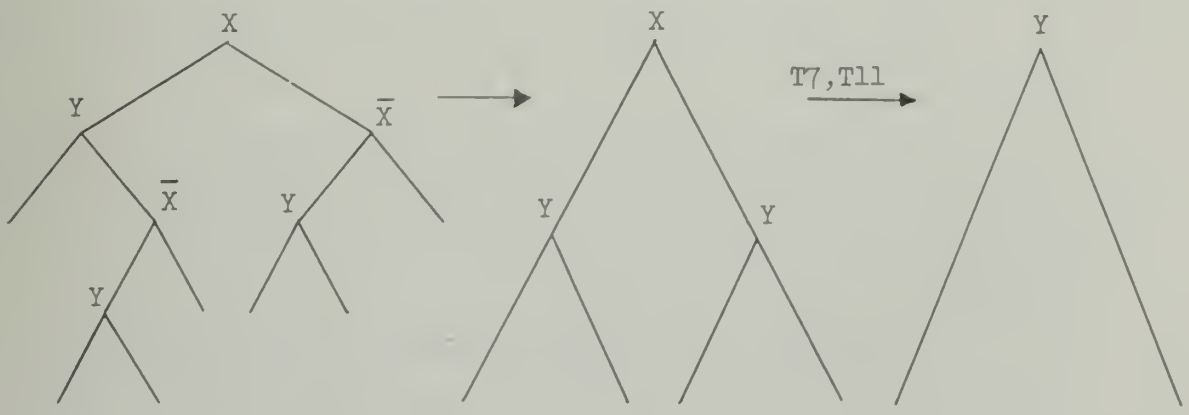
Boolean Algebra theorems which are useful in simplifying Boolean expressions can be used to eliminate nonexecutable paths.

Examples:

1. $X \vee XY = X$ is equivalent to

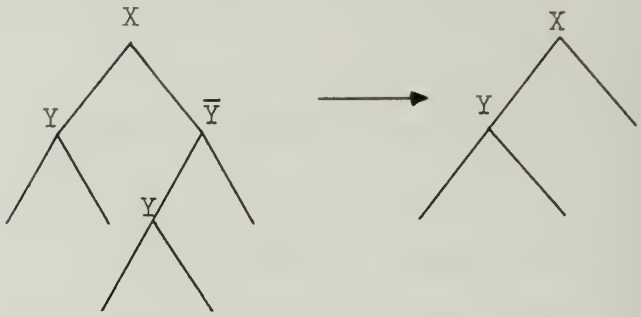


2. $XY \vee \bar{X}Y = Y$ is equivalent to the transformation

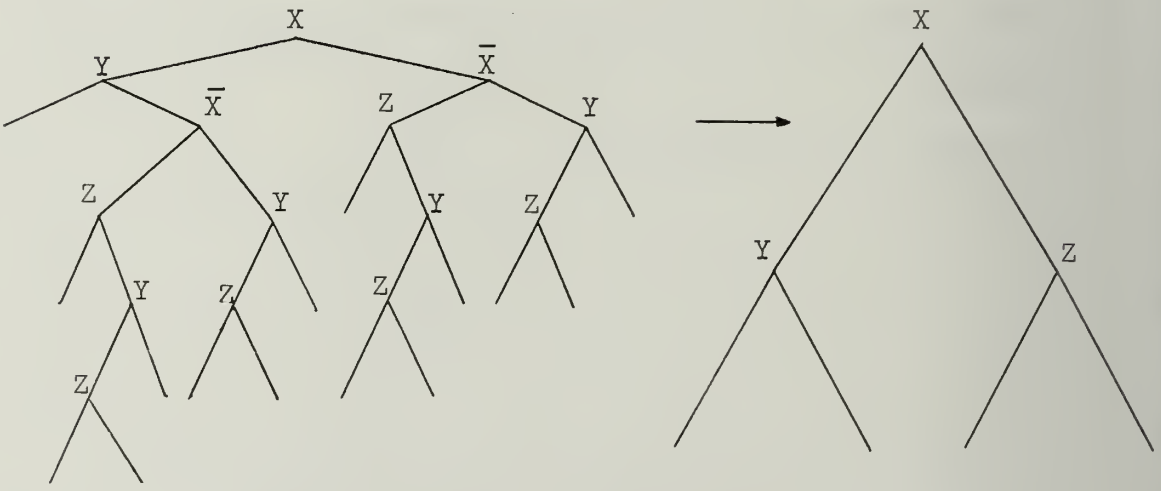


In the following stages of the optimization the topological transformations T11 and T7 will transform the tree on the right to a tree equivalent to the expression Y.

3. $(X \vee \bar{Y})Y = XY$ is equivalent to the transformation



4. $XY \vee \bar{X}Z \vee YZ = XY \vee \bar{X}Z$ is equivalent to



The transformations in 1) - 4) do not add statements to the program.

The following theorem will give a procedure for optimizing programs that have tests under reasonable cost functions.

THEOREM 15

There is an algorithm that finds an optimal program Π' for a given program Π that has tests. The algorithm operates in a series

of steps so that Π is first transformed by logical transformations to an equivalent program Π_0 that has elementary tests and negations only. Using logical transformations that consist of negations only, Π_0 is then transformed to a program Π_0'. Π' is obtained by operating the topological transformations on Π_0'.

PROOF:

We first assume that all nonexecutable paths can be eliminated from Π in such a way that statements are not added to the program. Thus in every optimal program each path is executable. Take Π_1 to be a program logically equivalent to Π in which each path is executable.

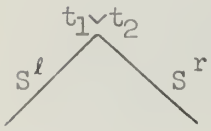
$$\Pi \xrightarrow[L_1]{} \Pi_1$$

where L_1 are the logical transformations necessary to transform Π into Π_1 which are similar to T6 but operate also on negations of tests.

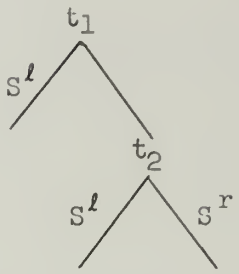
By the definition of reasonable cost functions every optimal program has elementary tests and negations of elementary tests only. That is because the number of tests performed is minimized if logical expressions are broken down into tests that do not have the $\vee$ and $\cdot$ operations. Take Π_0 to be a program logically equivalent to Π_1 which has minimum number of tests performed. Π_0 is obtained by breaking the logical expressions in Π_1 into elementary tests and negations of elementary tests. In some cases de Morgan's laws must be used so that breaking the logical functions would be possible. $\Pi_1 \xrightarrow[L_0]{} \Pi_0$ where L_0 is the set of logical transformations necessary to transform Π_1 into Π_0.

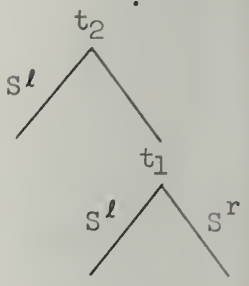
There may be several programs equivalent to Π_1 that have elementary tests and negations of elementary tests only. For example

is equivalent to



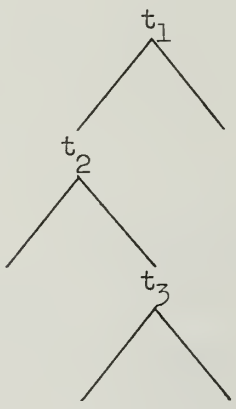
but also to



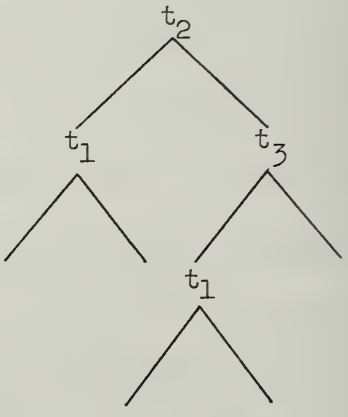


Thus the process of constructing Π_0 is nondeterministic. Only a finite number of Π_0 can be reached, because breaking the expressions can be done in a finite number of forms only.

In the above example the two possible trees have the same cost provided there is no additional information on the tests. But for some cases if we permute the terms we might reduce the cost of the program. For example $t_1(t_2 \vee t_3)$ and $(t_2 \vee t_3)t_1$ are the same functions according to the definition of Boolean algebra but the corresponding trees



$t_1(t_2 \vee t_3)$



$(t_2 \vee t_3)t_1$

are different and their costs are not the same. (The average number of tests performed by the program on the right is greater.)

In order to eliminate checking all possible programs with elementary tests equivalent to the given program, we would like to have an algorithm that given a Boolean expression will find the minimal tree corresponding to it. This problem is a special case of the problem presented in Slagle (10). That paper is concerned with finding a minimum-cost tree equivalent to the given Boolean expression where the cost of applying each elementary test and the probability of its outcome are given. When no information is given on the tests we may assume that the costs of the tests are equal and the probability of each test to be true is equal to its probability to be false. Thus we get a special case of the general problem.

The algorithm finds a low cost tree for the given expression in the following way:

Suppose an expression S is of the form $S=S_1 \vee \ldots \vee S_n$. Each permutation of the S_j gives a different tree. To obtain a low cost expression, expressions S_j that have low cost and high probability of being true should be carried out first. We will distinguish between the following cases:

a) for a variable x_i the expression that gives a low cost tree is x_i itself.

b) if an expression is of the form $S_1 \vee \ldots \vee S_n$ and none of S_j is a disjunction then find for each S_j a low cost expression R_j and the low cost expression for S will be a permutation of the R_j $R_1 \vee \ldots \vee R_n$ such that

$$\frac{c_1}{p_1} \leq \ldots\ldots\ldots \leq \frac{c_n}{p_n}$$

where c_j is the cost of R_j and p_j is its probability to be true.

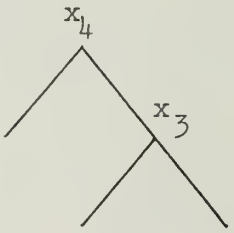
c) if the expression is of the form $S_1 \ldots\ldots S_j \ldots S_n$ and none of the S_j is a conjunction then find for each S_j a low cost expression R_j and then permute the R_j and get the expression $R_1 \ldots\ldots R_n$ such that

$$\frac{c_1}{q_1} \leq \ldots\ldots \leq \frac{c_n}{q_n}$$

where c_j is the cost of R_j and $q_j = 1-p_j$, p_j is its probability to be true.

Suppose we wish to find a low cost tree for $x_1 \vee x_2(x_3 \vee x_4)$ and c_i, p_i are as follows: $c_i = (176, 80, 40, 42)$, $p_i = (\frac{1}{2}, \frac{1}{2}, \frac{1}{4}, \frac{1}{2})$. We first find low cost expressions for x_1 and $x_2(x_3 \vee x_4)$. For x_1 we get x_1 itself. For $x_2(x_3 \vee x_4)$ we find low cost expressions for x_2 and $x_3 \vee x_4$. For $x_3 \vee x_4$ we have $\frac{42}{\frac{1}{2}} < \frac{40}{\frac{1}{4}}$ therefore the low cost expression is $x_4 \vee x_3$.

Its cost is

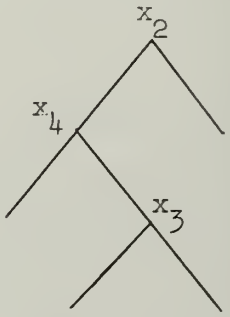


$$c_4 p_4 + (c_4 + c_3) q_4 p_3 + (c_4 + c_3) q_4 q_3 = c_4 + q_4 c_3 = 62.$$

Its probability is $p_4 + q_4 p_3 = \frac{5}{8}$.

The low cost expression for $x_2(x_3 \vee x_4)$ is $x_2(x_4 \vee x_3)$ because

$$\frac{c(x_2)}{q(x_2)} = \frac{80}{\frac{1}{2}} < \frac{c(x_4 \vee x_3)}{q(x_4 \vee x_3)} = \frac{62}{\frac{3}{8}}.$$



The cost of

is 111 and its probability is $\frac{5}{16}$.

The low cost expression for $x_1 \vee x_2(x_3 \vee x_4)$ is $x_1 \vee x_2(x_4 \vee x_3)$ because $\frac{176}{\frac{1}{2}} < \frac{111}{\frac{5}{16}}$.

If no information is given on the tests, i.e. we assume $c_i = 1 \quad p_i = \frac{1}{2}$ for all i, then the low cost expression for $x_2(x_3 \vee x_4)$ can be either $x_2(x_3 \vee x_4)$ or $x_2(x_4 \vee x_3)$ but not $(x_3 \vee x_4)x_2$ because

$$\frac{1}{\frac{1}{2}} = \frac{c(x_2)}{q(x_2)} < \frac{c(x_3 \vee x_4)}{q(x_3 \vee x_4)} = \frac{1\frac{1}{2}}{\frac{1}{4}}$$

For the case of disjunctions of conjunctions of singly occuring variables and for the case of conjunctions of disjunctions of singly occuring variables the algorithm above is proved to find a minimum cost tree. Although this fact cannot be extended to the general case, an exhaustive search in the general case may be made more efficient by using the above algorithm.

Several improvements can be made to Slagle's algorithm concerning extending his results to variables appearing more than once:

i) as we showed above some cases of variables occuring more than once can be eliminated by getting rid of nonexecutable paths. This holds for

$$X_1 \ldots X_n \vee X_1 \ldots X_n Y_1 \ldots Y_m = X_1 \ldots X_n$$

we eliminated the double occurance of $X_1 \ldots X_n$. This is done by eliminating nonexecutable paths of the program. This is also true for

$$X_1 \ldots X_n (X_1 \ldots X_n \vee Y_1 \ldots Y_m) = X_1 \ldots X_n$$

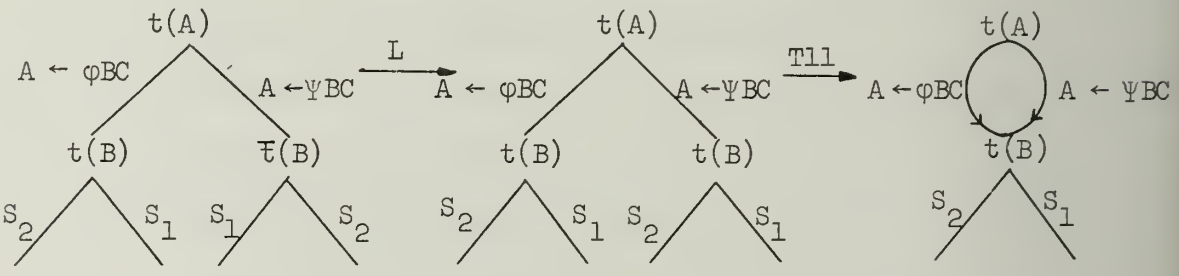
$$X Y_1 \ldots Y_n \vee \bar{X} Y_1 \ldots Y_n = Y_1 \ldots Y_n$$

ii) the algorithm applies also to those cases of disjunctions of conjunctions of variables appearing more than once that can be expressed as conjunctions of disjunctions. (e.g. $t_2t_1 \vee t_1t_3 = t_1(t_2 \vee t_3)$).

We might conclude that the algorithm above might be used to get Π_0 quicker, either in its general form when the cost of applying each elementary test and the probability of its outcome are given, or in the special case of the algorithm when no information is given.

$\Pi \xrightarrow[L_1 \cup L_0]{} \Pi_0$ $\quad \Pi \equiv \Pi_0$. We define $L = L_1 \cup L_0$.

Π_0 and Π' both have elementary tests and negations only, and are equivalent by a set of transformations composed of topological transformations and logical transformations that consist of negations only. Logical transformations that include negations might expose identical subgraphs. Example:



Also logical transformations that include negations only might make flipping of tests possible. Therefore we will transform Π_0 to a program Π_0' equivalent to Π_0 by logical transformations only, such that Π_0' is equivalent to Π' by topological transformations only. If Π' is not known all choices for Π_0' must be considered, thus Π_0' is constructed nondeterministically. There is only a finite number of possible Π_0'

but some algorithm or heuristics based on the nature of the cost function might get Π_0' quicker. (In some cases all Π_0' have the same cost, but the costs might be different for some cost functions. For example, the cost of the test for zero might be less than the cost of the test for a non-zero variable.) A good algorithm will construct Π_0' such that tests checking the same values will have the same logical value, so that applications of T8 and T11 (if needed) might be possible.

$$\Pi_0 \xrightarrow[L']{} \Pi_0'$$

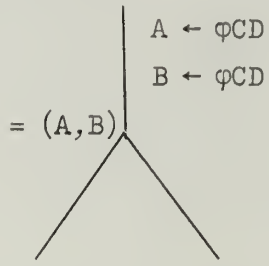
$$\Pi_0' \xrightarrow[\{1,2,3,4,5,7,8,9,10,11\}]{} \Pi'$$

The algorithm to obtain an optimal program Π' from Π_0' by using topological transformations only is the same as in Theorem 10.

Since we showed that breaking down the tests into elementary tests reduces the cost of the program, a different algorithm from that of Theorem 15 could be designed which first breaks down the tests as to get some equivalent program with elementary tests only, and then operates the topological transformations in order to get an optimal program. The algorithm of Theorem 10 cannot be used here because it does not find a program with the minimum number of elementary tests. Obtaining an optimal program with the minimum number of tests by using topological transformations only is more complicated than Slagle's algorithm since it involves a process of adding new tests and new nonexecutable paths to the program and eliminating other tests that become useless and paths that become nonexecutable.

We showed how Boolean manipulations of the tests eliminated nonexecutable paths and therefore reduced the cost of the program.

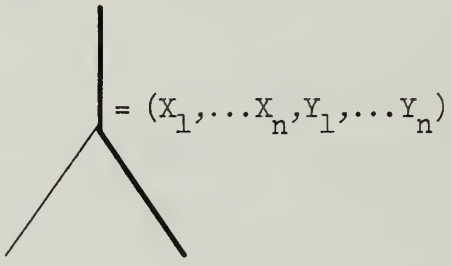
Optimizing the program by eliminating nonexecutable paths can also be done when we have additional information on the tests. As an example, we will show how the use of the test for equality may eliminate nonexecutable paths. If we have the program



$v(A) = v(B)$ and the right path can never be executed.

This case is similar to the case in which algebraic identities are known to hold among operators. Here the additional information is on the tests. Here the model of program schemata is as before, only test statements include tests instead of test names. The following cases are possible:

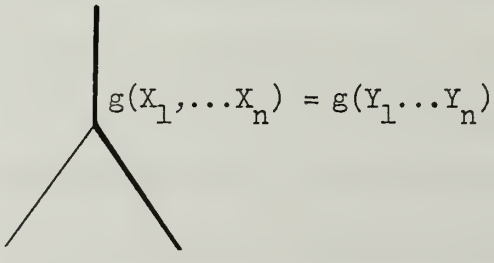
(i)



$v^{\ell}(X_i) = v^{\ell}(Y_i) \qquad 1 \leq i \leq n.$

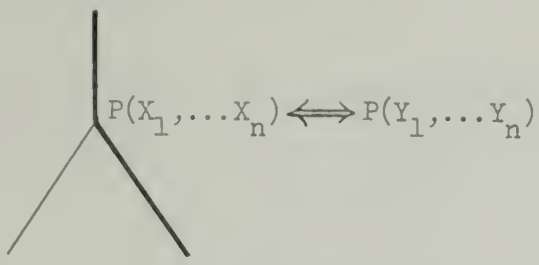
The marked path is nonexecutable and can be eliminated.

(ii)



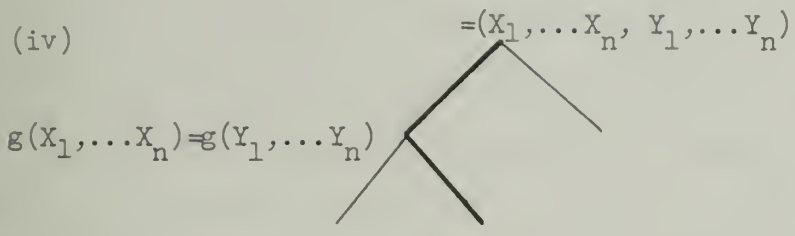
where $v^{\ell}(X_i) = v^{\ell}(Y_i)$ $1 \leq i \leq n$, and g is any function. The marked path is nonexecutable.

(iii)



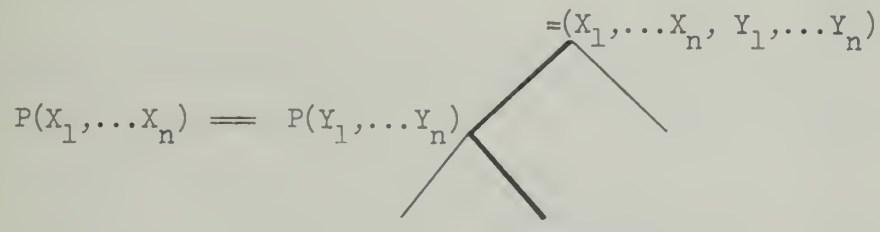
$v^l(X_i) = v^l(Y_i)$ $1 \leq i \leq n$, and P is a proposition.

(iv)



g is any function. Here the values of X_i and Y_i might be different for some i (i.e. the expressions are different) but the vectors (X_i) and (Y_i) might be equal under some interpretation. The marked path is nonexecutable.

(v)



P is a proposition.

All these transformations preserve equivalence of programs. A scheme for optimization similar to that of Theorem 15 can be constructed for the cases above.

9. PROGRAM SCHEMATA WHICH ALWAYS HALT

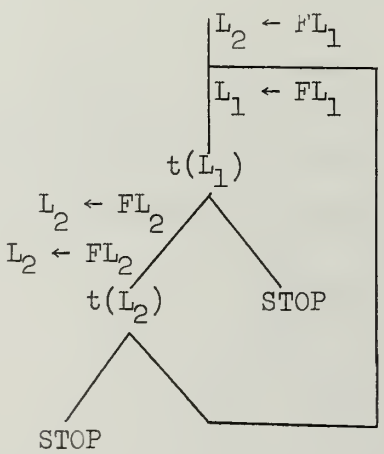
The equivalence problem for the class of program schemata which always halt is decidable, although membership in this class is not. (See Luckham, Park and Paterson (8)).

The decision procedure of (8) is based on the fact that for each program that halts under all interpretations we can construct effectively an equivalent loop-free schema. We unwind the loops, discarding any paths which are nonexecutable. If this process continues indefinitely, the hypotheses of Koenig's lemma are satisfied, implying that there is an infinite executable path and thus the program diverges under some interpretation. Therefore for programs which always halt this procedure must produce an equivalent, finite, loop-free program. Thus the decision procedure of Theorem 2 applies also to programs which always halt.

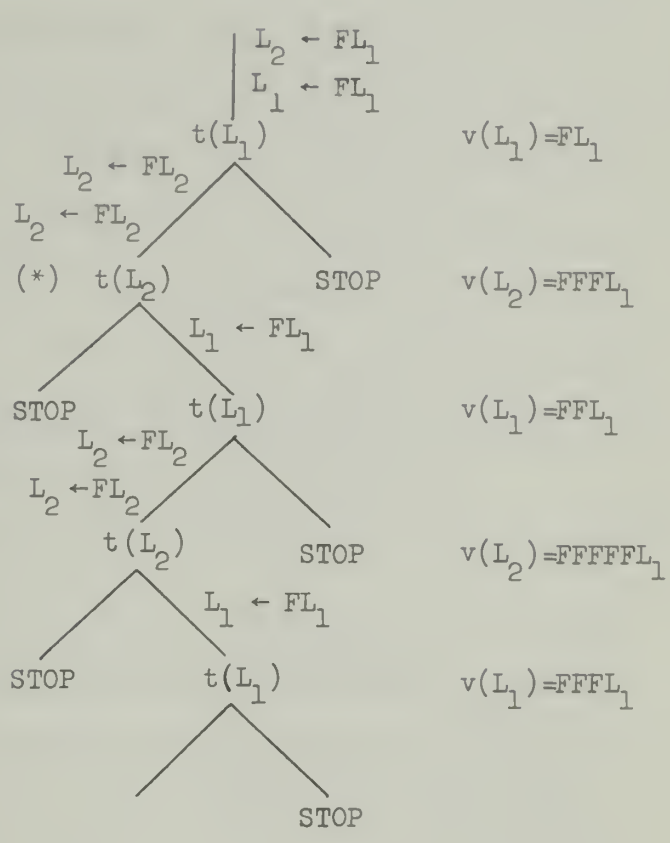
EXAMPLE 20

Produce an equivalent loop-free program for a program which always halts.

Π:



Π' is an equivalent loop-free program:



The last test statement $t(L_1)$ tests the same value as does the statement marked by (*). Therefore the left path is nonexecutable and the procedure stops. We get an equivalent loop-free program.

All the transformations that operate on loop-free programs can be used to simplify programs which always halt. We will denote by T12 the transformation that maps programs which always halt to loop-free programs. Because T12 eliminates nonexecutable paths, every program which always halts is transformed by T12 to a loop-free program in which each path is executable.

THEOREM 16

Let Π and Π' be two programs which always halt. Then

$$\Pi \xrightarrow[\{1,2,6,7,9,10,11,12\}]{*} \Pi' \quad .$$

The proof is obvious by using the results of the previous chapters.

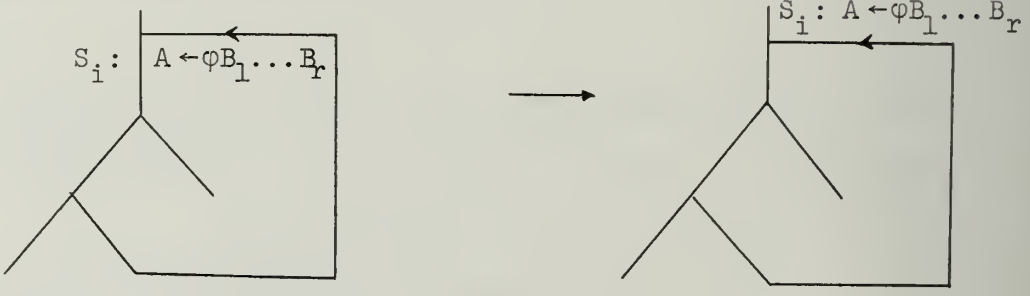
By using T12 first, and then applying the procedure of Theorem 10 for optimizing loop-free programs, we might optimize programs which always halt, under reasonable cost functions. The following transformations that operate on programs with loops only, and are equivalent to sequences of transformations of T1-T12, might be used by the optimizing process.

T13 Moving loop-independent statements out of the loop

Let S_i: $A \leftarrow \varphi B_1 \ldots B_r$ be a statement in a loop such that $B_1, \ldots, B_r$ are not defined by any statement in the loop. Then

(1) if A is not referenced by any statement in the loop S_i can be moved out of the loop either backwards or forwards.

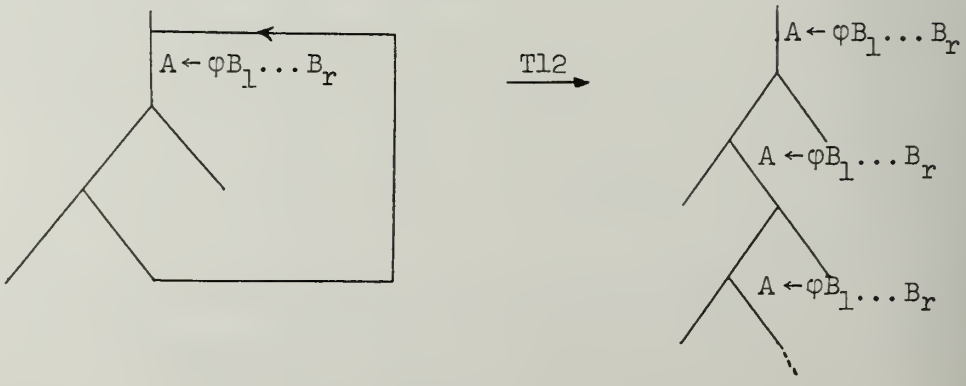
(2) if A is referenced by some statement S_j in the loop S_i can be moved backwards out of the loop.



PROPOSITION 1

If $\Pi \xrightarrow[(13)]{} \Pi'$, then $\Pi \xrightarrow[(1,2,4,12)]{*} \Pi'$.

PROOF:



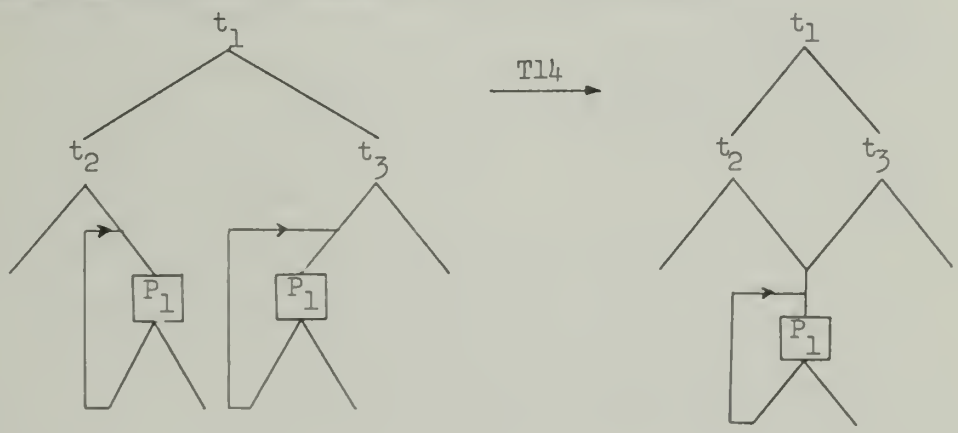
By T12 we get a corresponding loop-free program.

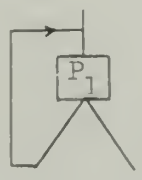
(1) If A is not referenced by any statement in the loop, all the statements except the last are useless and can be eliminated by several applications of T1. The last statement can be flipped forwards by applications of T4 so that it will be moved out of the loop.

(2) If A is either referenced or not by statements in the loop, we might use T2 several times to remove the redundant definitions of A, so that only the first definition is remained. Then we use T4 to flip the first statement backwards to move it out of the loop.

The reverse of T12 is used to transfer the loop-free program back to a loop program.

T14 Merging of identical subgraphs which include loops



We assume that the loop  always halts.

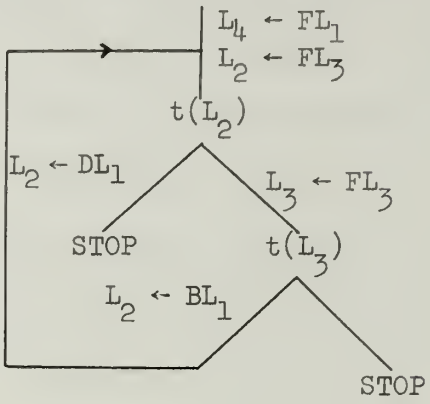
PROPOSITION 2

If $\Pi \xrightarrow[\{14\}]{} \Pi'$, then $\Pi \overset{*}{\xrightarrow[\{11,12\}]{}} \Pi'$.

PROOF:

T12 transforms Π to a loop-free program, then T11 merges the identical subgraphs and the reverse of T12 transforms the loop-free program back to a loop program.

T15 Elimination of nonexecutable loops



The loop can never be executed.

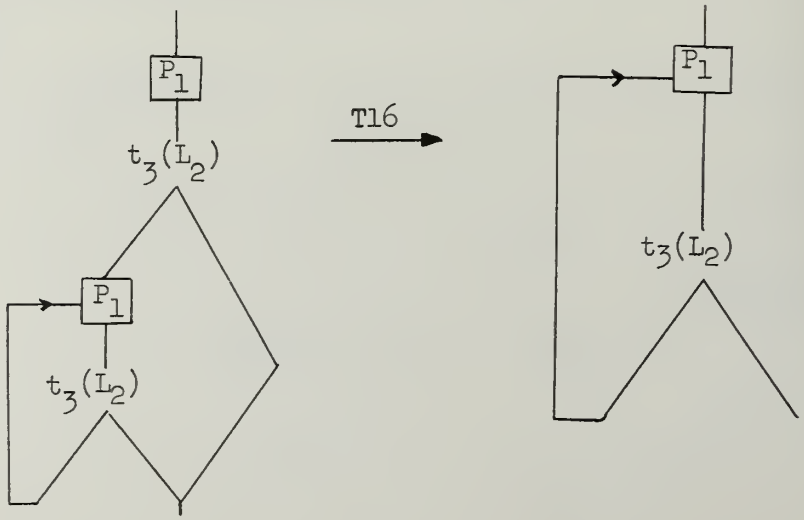
PROPOSITION 3

$$\text{If } \Pi \underset{\{15\}}{\Longrightarrow} \Pi', \quad \text{then} \quad \Pi \underset{\{12\}}{\Longrightarrow} \Pi' .$$

PROOF:

The procedure that transforms a program which always halts to a loop-free program, eliminates nonexecutable paths.

T16 Merging of loop-free code with loops



The two programs give the same loop-free program. The program on the right is more efficient.

PROPOSITION 4

$$\text{If } \Pi \underset{16}{\Longrightarrow} \Pi', \quad \text{then} \quad \Pi \overset{*}{\underset{12}{\Longrightarrow}} \Pi' .$$

PROOF:

T12 transforms Π to a loop-free program, which is transformed to Π' by the reverse of T12.

The optimizing process uses T13-T16 together with the transformations T1-T11.

T13 might be used to reduce the program.

T15 is used in the stage where nonexecutable paths are eliminated from the program.

T14 is used whenever T11 is used to merge subgraphs of the program. In this stage we can also apply T16 to merge as many loop-free codes with loops as possible.

REFERENCES

(1) A. V. Aho, and J. D. Ullman, "Transformations on Straight Line Programs," Conf. Record Second Annual ACM Symposium on Theory of Computing, pp. 136-148 (May 1970).

(2) F. E. Allen, "Program Optimization," Annual Review in Automatic Programming, Vol. 5 (1969).

(3) J. T. Baywell, "Local Optimizations," Proc. of a Symposium on Compiler Optimization, pp. 52-66 (July 1970).

(4) H. D. Huskey, and W. H. Wattenburg, "Compiling Techniques for Boolean Expressions and Conditional Statements in ALGOL 60," Comm. of the ACM, Vol. 4, pp. 70-75 (1961).

(5) IBM System/360 Operating System FORTRAN IV Programmer's Guide, SLR Number C28-6817.

(6) D. E. Knuth, The Art of Computer Programming, Vol. 1, Addison-Wesley Publishing Co. (1969).

(7) E. S. Lowry, and C. W. Medlock, "Object Code Optimization," Comm. of the ACM, Vol. 12, pp. 13-22 (1969).

(8) D. C. Luckham, D. M. R. Park, and M. S. Paterson, "On Formalized Computer Programs," J. Comp. and System Sciences, Vol. 4, pp. 220-249 (1970).

(9) M. S. Paterson, "Program Schemata," Machine Intelligence, Vol. 3, pp. 19-32 (1968).

(10) J. R. Slagle, "An Efficient Algorithm for Finding Certain Minimum-Cost Procedures for Making Binary Decisions," J. of the ACM, Vol. 11, pp. 253-264 (1964).

VITA

The author, Nurit Bracha, was born in Tel-Aviv, Israel. She received her B.A. degree in Mathematics and Statistics from the Hebrew University of Jerusalem in 1965, and her M.S. degree in Computer Science from the University of Illinois in 1970.

From September 1968 to January 1972, she worked as a research assistant in the Computing Services Office of the University of Illinois. She is currently a research associate with the Department of Computer Science and the Computer-based Education Research Laboratory, University of Illinois.

She is a member of Sigma Xi.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-72-516	2.	3. Recipient's Accession No.

4. Title and Subtitle

TRANSFORMATIONS ON LOOP-FREE PROGRAM SCHEMATA

5. Report Date: June, 1972

6.

7. Author(s): Nurit Bracha

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

A formal theoretical approach toward code optimization is considered. A program schema that models loop-free programs is presented and a complete set of equivalence preserving transformations on loop-free programs is found. A scheme for optimization is provided in which a sequence of these transformations is applied to get an optimal code. These results are extended to the model of loop-free programs which assumes that certain types of algebraic laws hold among the operators, and also to the case in which the tests are Boolean functions of elementary tests.

17. Key Words and Document Analysis. 17o. Descriptors

Program schemata, loop-free program, equivalence problem, transformations, optimization.

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Unlimited	UNCLASSIFIED	126
	20. Security Class (This Page) UNCLASSIFIED	22. Price

JUL 2 6 1972

BOUND BY
HNM

UNIVERSITY OF ILLINOIS-URBANA
510.84 ILGR no. C002 no.511-516(1972
RASER: random to serial converter.
3 0112 088400368